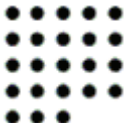


Entwicklung automatischer Installationsroutinen für Services auf Basis von JavaEE

Diplomarbeit

Vorgelegt an der Fachhochschule Köln

Campus Gummersbach

Fakultät 10

Institut für Informatik

Im Studiengang Allgemeine Informatik

ausgearbeitet von:

Markus-Alexander Müller

11043150

Erster Prüfer: Prof. Dr. Heide Faeskorn-Woyke

Zweiter Prüfer: Dipl.-Ing. Elektrotechnik Dejan Hofmann-Stajic

Gummersbach, im Mai 2010

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
ATP	Advanced Tool Plattform
CI	Configuration Item
EDV	Elektronische Datenverarbeitung
EJB	Enterprise Java Beans
JavaEE	Java Enterprise Edition
JAXB	Java Architecture for XML Binding
JDBC	Java Database Connectivity
JMS	Java Message Service
JMX	Java Management Extension
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSR	Java Specification Request
NGN	Next Generation Network
SAX	Simple API for XML
SOA	Service Oriented Architecture
SQL	Structured Query Language
UML	Unified Modeling Language
VPN	Virtual Private Network
XML	Extensible Markup Language

1. Einleitung

„Das Ganze ist mehr als die Summe seiner Teile“

(Aristoteles)

Wiederverwendbarkeit von Komponenten prägt schon seit einigen Jahren die Entwicklung in der Informatik. Während man diese bei den Programmiersprachen in der Objektorientierung durch das Konzept der Klassen wiederfindet, haben sich auf Business Ebene vor allem die sogenannten „Services“ etabliert.

Services kapseln die Geschäftslogik in kleinere wieder verwendbare Einzelteile. Ein Service muss dabei jedoch nicht für sich alleine stehen. Zwar ist der von ihm bereitgestellte Teilaspekt in sich gekapselt, jedoch ist es möglich mehrere Services zu kombinieren, um einen neuen Service zu bilden.

Dies soll exemplarisch verdeutlicht werden. Angenommen es gibt einen Service zum Überprüfen von Bankdaten, einen zum Verschicken von Waren und zuletzt einen zum Erstellen von Rechnungen. Jeder dieser Services ist in sich gekapselt und kann damit unabhängig für sich arbeiten. Aus diesen drei Services soll nun ein neuer Service entstehen, welcher einen Bestellvorgang abbilden soll. Dieser greift, soweit möglich, auf bereits vorhandene Komponenten zu. So würde der Service *Bestellvorgang* beispielsweise die drei bereits vorhandenen Services für *Rechnungen erstellen*, *Waren versenden* und *Bankdaten überprüfen* benutzen, anstatt diese Aspekte selber neu zu implementieren. Es wird also ein komplexerer Service aus den drei anderen Services gebildet. (Abbildung 1-1)

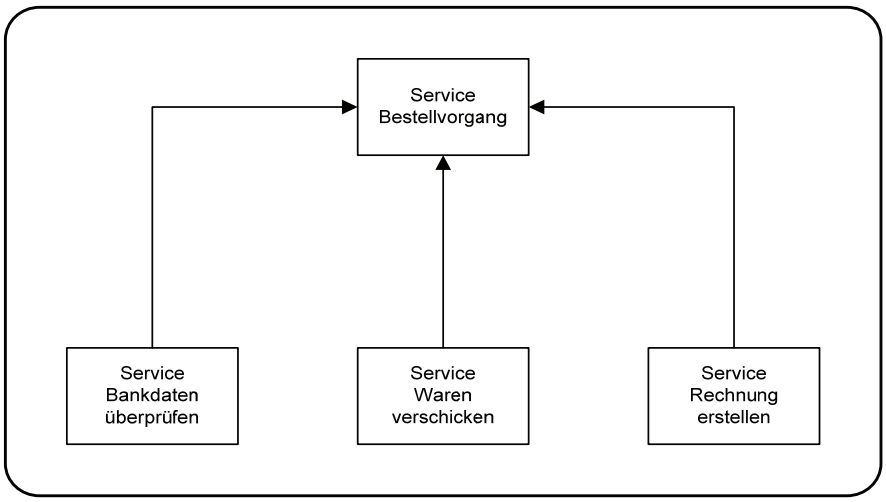


Abbildung 1-1: Zusammenstellen von Services

Hierbei wird genau wie bei Klassen jeder Service für sich als Blackbox angesehen. Dies bedeutet, dass von einem Service lediglich seine Schnittstellen bekannt sind. Was innerhalb des Services passiert, ist für den Benutzer oder den aufrufenden Service nicht zu erkennen.

Durch das Zusammenfügen von Services zu einem neuen kompletten Paket entstehen jedoch auch Abhängigkeiten zwischen diesen Services, da sich diese gegenseitig benutzen, beziehungsweise Daten voneinander benötigen. So könnte in diesem Beispiel ein Bestellvorgang gar nicht ausgeführt werden, wenn der Service *Bankdaten überprüfen* nicht zur Verfügung steht, oder er nicht durch seine Rückgabe bestätigt, dass die gegebenen Bankdaten korrekt sind.

Dies kann in großen Umgebungen schnell unübersichtlich werden. Außerdem sind Abhängigkeitsketten möglich, in denen ein abhängiger Service wiederum weitere Abhängigkeiten besitzt. Ziel dieser Diplomarbeit ist es, eine Methode zur automatischen und vereinfachten Installation von Services zur Verfügung zu stellen, indem Abhängigkeiten automatisch erkannt und aufgelöst werden. Des Weiteren sollen, soweit möglich, die Konfigurationsschritte, die bei einer Installation vorzunehmen sind, automatisiert werden. Dies beinhaltet das Anlegen von benötigten Ressourcen, die für den reibungslosen Betrieb eines Services benötigt werden, sowie eventuelle Anpassungen von Konfigurationsdateien eines Services. Als weitere Anforderung muss die Installation eines Services zu einem späteren Zeitpunkt nachvollziehbar sein. Dies bedeutet, dass eine Historisierung von Installationen eingeführt werden soll, an welcher ersichtlich ist, welcher Service wann und wo installiert wurde. Nach einer Installation soll also ein konsistenter Zustand erreicht sein, in dem ein Service mit seinen Abhängigkeiten installiert und konfiguriert ist, und somit einwandfrei funktioniert.

Diese Diplomarbeit fand in enger Zusammenarbeit mit der QSC AG in Köln statt, so dass deren Umgebung als konkretes Beispiel diente. Die QSC AG ist ein mittelständisches Unternehmen, das bundesweit NGN-Telekommunikationsdienste für Unternehmen jeder Größenordnung anbietet. Sie bietet unter anderem IP-Telefonie und Vernetzung von ganzen Standorten über VPN an. Hierbei greift sie auf ein eigenes Breitbandnetz zurück. Da Services bei der QSC sowohl für interne Prozesse, als auch für die Anbindung von externen Kunden eine große Rolle spielen, besteht großes Interesse an der Optimierung der Installation und Wartung von Services.

Um das Verständnis zu Einzelaspekten der Diplomarbeit zu erleichtern, sind die Kapitel jeweils inhaltlich geschlossen strukturiert und lassen sich sowohl einzeln, als auch in ihrem Zusammenhang betrachten.

In Kapitel 1 werden Grundlagen zu Services und Services unter JavaEE erklärt. Die Grundlagen umfassen dabei zwei Teilaspekte. Zum Einen sollen alle wichtigen Sachverhalte zum Verständnis der Probleme bei der Entwicklung von automatischen Installationsroutinen erläutert, zum Anderen die technischen Hintergründe, die zum Verständnis des entwickelten Prototypen nötig sind, vermittelt werden.

Kapitel 3 widmet sich den zu lösenden Problemen, um eine automatische Installation von Services zu ermöglichen. Hierbei wird zunächst das Vorgehen bei einer manuellen Installation erläutert, um hieraus abzuleiten, welche Teilaspekte es zu automatisieren gilt.

Kapitel 4 dient der Analyse verschiedener Implementierungsmöglichkeiten. Dort werden kommerzielle Lösungen, Anpassungen von Open-Source-Lösungen sowie eine Eigenentwicklung auf ihre Tauglichkeit geprüft und verglichen.

Ein Überblick über alle zur Entwicklung des Prototyps verwendeten Produkte wird in Kapitel 5 gegeben.

Kapitel 6 zeigt die Lösung der Problematik bei der automatischen Installation anhand der konkreten Implementierung eines Prototyps.

Abschließend wird in Kapitel 7 eine Einschätzung der künftigen Entwicklung, sowie ein Rückblick auf diese Arbeit dargelegt.

2. Grundlagen

Zum besseren Verständnis von Services und deren Konfiguration im Umfeld von JavaEE wird im Folgenden ein kurzer Überblick über die wichtigsten, für diese Arbeit ausschlaggebenden, Grundbegriffe gegeben. Alle in dieser Arbeit verwendeten Services wurden mit Hilfe von JavaEE implementiert, weshalb andere Möglichkeiten der Implementierung nicht betrachtet werden.

2.1. Services

Ein Service ist eine in sich gekapselte Komponente, die eine für bestimmte Aufgaben benötigte Funktionalität nach Außen zur Verfügung stellt. Um bei dem in der Einleitung vorgestellten Beispiel zu bleiben, kann ein Service beispielsweise das Überprüfen von Bankdaten zur Verfügung stellen. Dieser bekommt Bankdaten als Eingabeparameter übergeben und liefert das Ergebnis der Gültigkeitsprüfung zurück. Die Implementierung des Service wird immer als Blackbox betrachtet, lediglich das Ergebnis ist von Interesse.

Ziel bei der Gestaltung von Services ist es, EDV-Komponenten wie Datenbankabfragen oder Berechnungen sinnvoll zu kapseln, so dass ihre Zusammenführung der Geschäftslogik dient. Hierbei ähnelt die Kapselung vom Konzept den aus der Programmierung bekannten Klassen. Während diese jedoch meist technische Aufgaben kapseln, orientieren sich Services an Geschäftsprozessen und kapseln fachliche Aufgaben.

Ein wichtiger Aspekt in Zusammenhang mit Services ist das Zusammenfügen von mehreren Services zu einem neuen Service. Hierbei entsteht durch die Kombination von Services einer niedrigeren Ebene ein neuer Service der höheren Ebene. Aus diesem Grund ist es wichtig Services möglichst in sinnvolle fachliche Aufgaben zu partitionieren, damit die Wiederverwendbarkeit gewährleistet werden kann.

2.2. Java Enterprise Edition (JaveEE)

JavaEE steht für Java Enterprise Edition. JavaEE ist genaugenommen kein alleinstehendes Produkt, sondern eine Sammlung von Spezifikationen, die zur Entwicklung und Ausführung von Enterprise Applikationen benötigt werden.

Hierbei stehen Merkmale im Vordergrund, die für den Enterprise Einsatz besonders wichtig sind. Hierzu zählen unter anderem die transaktionsbasierte Ausführung von Anwendungen, aber auch Merkmale wie Skalierbarkeit, Ausfallsicherheit und Interoperabilität.

Die in der Spezifikation definierten Softwarekomponenten und Dienste dienen dazu, einen allgemein akzeptierten Rahmen zu schaffen, in dem verteilte, mehrschichtige Anwendungen entwickelt werden können. Um dies zu erreichen, kann auf die in der Spezifikation definierten Komponenten zurückgegriffen werden. Klar definierte Schnittstellen zwischen den Komponenten sollen dafür sorgen, dass auch Softwarekomponenten von unterschiedlichen Herstellern zusammenarbeiten können.

Die Bestandteile der Spezifikation werden innerhalb des Java Community Process[1] von diversen Unternehmen erarbeitet. Im Anschluss wird diese Spezifikation als Dokument und Referenzimplementierung der Öffentlichkeit zur Verfügung gestellt.

JavaEE definiert somit eine standardisierte Middleware Plattform, die die Entwicklung von Enterprise Applikationen unterstützen soll. Hierbei werden dem Entwickler viele standardisierte Komponenten und Dienste zur Verfügung gestellt, die es ihm ermöglichen, sich auf die Entwicklung der Kernfunktionalität seiner Anwendung zu konzentrieren. Dies geschieht dadurch, dass häufig benötigte Funktionalitäten wie Persistierung oder Transaktionen bereits Bestandteil der JavaEE Spezifikation sind. Somit können sie auf standardisierte Weise benutzt werden, und müssen nicht eigenständig entwickelt werden.

2.3. JavaEE Applikationsserver

JavaEE-Komponenten benötigen eine spezielle Infrastruktur als Laufzeitumgebung. Diese Infrastruktur nennt sich JavaEE Applikationsserver, im Folgenden Applikationsserver genannt.

Ein Applikationsserver stellt die in der JavaEE Spezifikation beschriebenen Dienste und Komponenten in einer konkreten Implementierung zur Verfügung. Somit laufen Anwendungen, die mit JavaEE entwickelt wurden, nur auf einem Applikationsserver.

Ein JavaEE Applikationsserver wird in logische Systeme, die sogenannten Container, unterteilt.

Bei Containern handelt es sich um ein in der Softwareentwicklung verwendetes Entwurfsmuster. Dieses besagt, dass ein Container ein Behälter für von ihm definierte Komponenten ist. Diese Komponenten kapseln eine logische Information oder Anwendung des Systems und können zur Laufzeit verändert werden oder mit anderen Komponenten interagieren. Der Container hat die Aufgabe, die Schnittstellen zu dieser Interaktion mit

[1] Der Java Community Process ist ein 1998 eingerichtetes Verfahren welches zur Weiterentwicklung von Java und der Java Standard Bibliotheken angewendet wird; http://jcp.org

dem Rest des Systems zur Verfügung zu stellen und eventuelle Anfragen an eine von ihm verwaltete Komponente weiterzuleiten. Außerdem kann er weitere benötigte Dienste oder Funktionen für seine Komponenten zur Verfügung stellen. Ein wichtiges Merkmal von Containern ist die Möglichkeit, ihnen zur Laufzeit weitere Komponenten hinzuzufügen oder zu entfernen.

Ein Applikationsserver ist selbst ein Container, in dem mehrere andere Container oder Dienste als Komponenten laufen können. Jeder dieser Container stellt wiederum eine Laufzeitumgebung für bestimmte Komponenten dar.

Einige der gängigsten Vertreter sind:

- Ein Webserver-Container als Laufzeitumgebung für dynamische Webseiten, wie zum Beispiel JavaServerPages[2]
- Ein Servlet-Container als Laufzeitumgebung für Servlets[3]
- Ein Enterprise Java Bean(EJB)-Container als Laufzeitumgebung für Enterprise Java Beans[4].

Durch den modularen Aufbau des Applikationsservers mittels Container ist es möglich, diesem neue Funktionen zuzufügen.

In Abbildung 2-1 wird ein Applikationsserver mit einigen der spezifizierten Containern und Diensten schematisch dargestellt.

[2] JavaServerPages ist eine von Sun Microsystems entwickelte auf JHTML basierende Web-Programmiersprache zur einfachen dynamsichen erzeugung von HTML- und XML-Ausgaben eines Webservers. http://java.sun.com/products/jsp/index.jsp

[3] Servlets setzt sich aus den Worten Server und den von Java bekannten Applet zusammen und ist somit ein serverseitig laufendes Applet. http://java.sun.com/products/servlet/index.jsp

[4] Siehe Kapitel 2.5

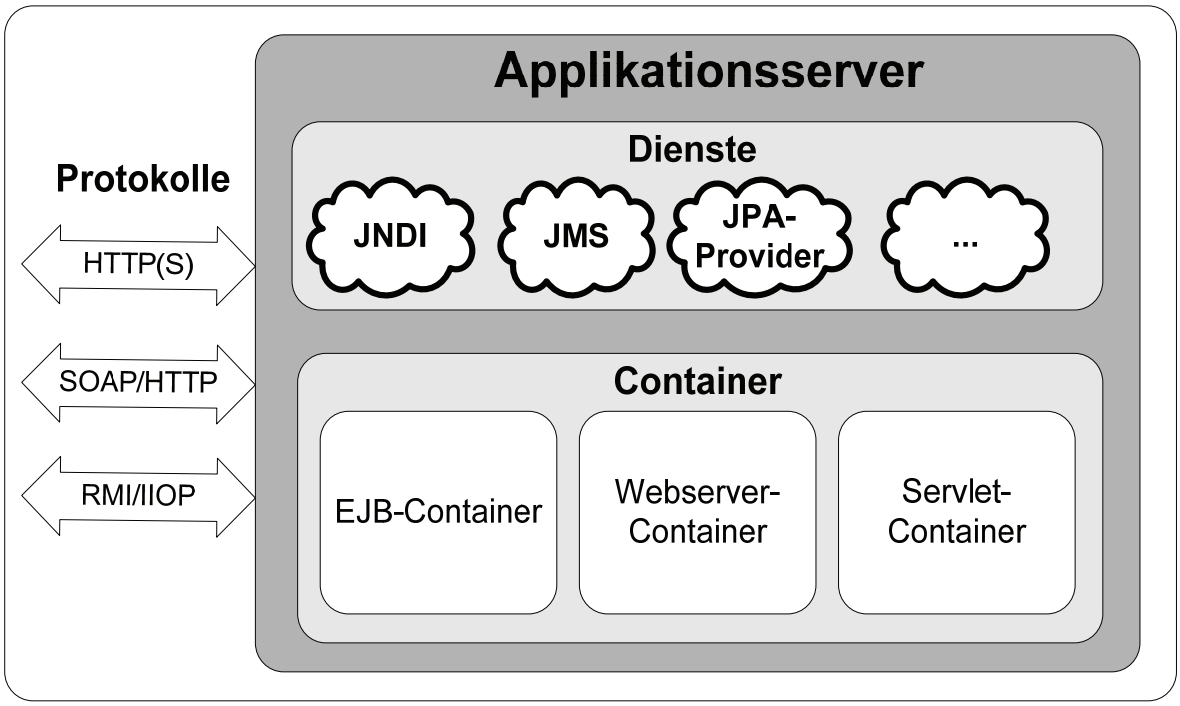


Abbildung 2-1: Schematische Darstellung eines Applikationsservers

Zusammenfassend ist ein Applikationsserver die Implementierung der von JavaEE geforderten Spezifikationen und stellt somit die Laufzeitumgebung für Enterprise Anwendungen, die mit Hilfe von JavaEE entwickelt wurden, dar.

2.4. Services unter JavaEE

Wie bereits erwähnt, handelt es sich bei Services um Komponenten, die einzelne funktionale Themen – meist Geschäftslogik – kapseln. Unter JavaEE werden diese gekapselten Komponenten in der Regel durch Enterprise Java Beans (EJBs) dargestellt. EJBs sind in Java geschriebene Komponenten, welche die fachliche Logik eines Services implementieren. Ihre Laufzeitumgebung ist ein EJB-Container innerhalb eines Applikationsservers. Die von ihnen bereitgestellten Service-Methoden werden durch ein normales Java-Interface definiert, welches die Komponente implementiert. Dieses Interface wird auch als „Business Interface“ bezeichnet und stellt die von Außen zugängliche Schnittstelle dar.

Wird ein Service über sein Interface aufgerufen, so ist der EJB-Container dafür verantwortlich, die Anfrage an eine entsprechende EJB-Komponente weiterzuleiten, beziehungsweise eine solche Komponente zu instanziieren. Dies bedeutet, dass jegliche infrastrukturellen Aufgaben vom Container übernommen werden, so dass sich der Entwickler einer EJB auf ihre fachlichen Aufgaben konzentrieren kann. In Abbildung 2-2 ist der EJB-Container, so wie der Aufruf einer EJB schematisch dargestellt.

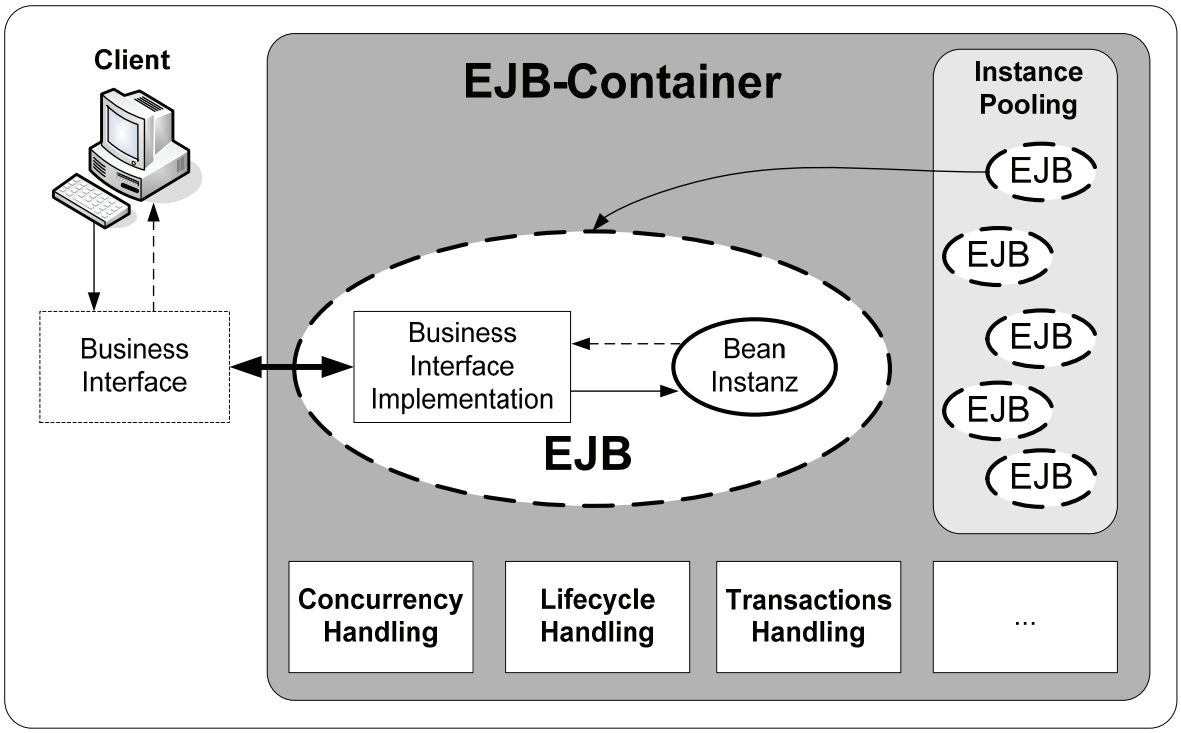


Abbildung 2-2: Schematischer Aufbau eines EJB-Containers

2.5. Enterprise JavaBeans (EJB)

EJBs stellen die Implementierung von Services unter JavaEE dar. Sie sind Teil der Spezifikation von JavaEE und liegen zu diesem Zeitpunkt in der Version 3.0 vor. In der EJB Spezifikation werden drei verschiedene Typen von Beans unterschieden, welche im Folgenden kurz vorgestellt werden.

2.5.1. Session Beans

Die im Zusammenhang mit Services wohl wichtigsten EJBs sind die Session Beans. Die Session Beans bilden die Logik der Geschäftsprozesse ab und stellen somit den eigentlichen Kern eines Services dar. Innerhalb der Session Beans wird zwischen den zustandsbehafteten (stateful) und zustandslosen (stateless) Beans unterschieden. Die Ausprägungen kennzeichnen hierbei die Dauerhaftigkeit der Verbindung einer Session Bean zu einem Client. Eine „Stateful Session Bean“ bleibt ihrem Client so lange zugeordnet, bis dieser sie freigibt. Dies bedeutet, dass in der Zeit dieser Zuordnung kein anderer Client mit dieser Bean kommunizieren kann. Eine „Stateful Session Bean“ ist in der Lage, die von einem Client übergebenen Daten auch über mehrere Methodenaufrufe hinweg intern zu speichern. „Stateless Session Beans“ hingegen sind nur für die Dauer eines Methodenaufrufs ihrem Client zugeordnet. Somit können sie nach der Abarbeitung einer Methode sofort wieder einem anderen Client zur Verfügung stehen. Aufgrund dieser

kurzen Bindung ist es ihr auch nicht möglich, von ihrem Client übermittelte Daten intern zu speichern.

Die zur Verfügung stehenden Methoden werden von einer Session Bean durch ihr Business-Interface definiert. Dieses Interface ist die Repräsentation nach außen. Ein Client kennt nur dieses Interface und kann somit nur die hier definierten Methoden aufrufen.

2.5.2. Message-Driven Beans

Eine weitere Art der Enterprise Java Beans sind die „Message Driven Beans“. Sie dienen der asynchronen Kommunikation über den Java Messaging Service[5] (JMS). Sie werden häufig zur Anbindung bestehender Dritt-Systeme (Legacy-Kompatibilität) verwendet. Grundsätzlich ähnelt eine Message-Driven Bean einer Stateless Session Bean. Message-Driven Beans können jedoch nicht direkt von einem Client aus angesprochen werden, sondern lediglich indirekt über Nachrichten.

Sie dienen dabei als Nachrichten-Endpunkte. Sie werden benachrichtigt, wenn eine für sie bestimmte Nachricht eintrifft und werden dadurch aktiviert.

2.5.3. Persistent Entity

Als letzte Ausprägung gibt es noch die Persistent Entities. Sie modellieren die persistenten Daten des Systems und repräsentieren somit zum Beispiel einen Datensatz aus einer Datenbank.

Zum Mapping dieser Persistent Entity auf eine relationale Datenbank wird die Java Persistence API[6] (JPA) eingesetzt.

2.6. Deployment Descriptor

Der Deployment Descriptor stellt die Einsatzbeschreibung einer EJB dar. Es handelt sich um eine XML Konfigurationsdatei, in welcher beschrieben wird, welche Ressourcen eine EJB benötigt, wie diese heißt, welche anderen EJBs diese aufrufen kann, sowie weitere Merkmale die für den Einsatz nötig sind. Mit Einführung von der EJB 3.0 Spezifikation hat der Deployment Descriptor etwas an Bedeutung verloren, da viele dieser Einstellungen nun vorzugsweise als Annotationen[7] innerhalb des Quellcodes angegeben werden. Jedoch sind weiterhin beide Wege möglich, wobei Definitionen im Deployment Descriptor die

[5] Siehe Kapitel 2.9
[6] Siehe Kapitel 2.8
[7] Siehe Kapitel 2.7

Einstellungen der Annotationen überschreiben. Weiterhin ist der Deployment Descriptor eine zentrale Datei, welche zur Konfiguration von EJBs genutzt werden kann. Abbildung 2-3 zeigt ein Beispiel für einen Deployment Descriptor.

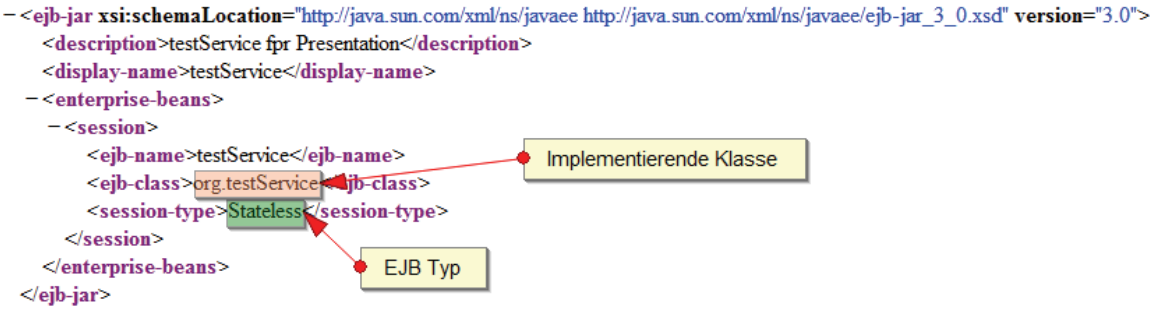


Abbildung 2-3: Beispiel eines Deployment Descriptors

2.7. Annotationen

Annotationen wurden mit Java in der Version 5 eingeführt. Sie werden innerhalb des Quellcodes angegeben und beginnen mit einem „@“ Zeichen. Sie ermöglichen die Angabe von Metadaten innerhalb des Quellcodes. Häufige Verwendung finden diese im JavaEE Umfeld.

Durch Annotationen ist es möglich, Meta-Informationen direkt im Quellcode zu definieren. So verwenden Enterprise Java Beans seit Version 3.0 zu einem Großteil Annotationen, anstatt des teilweise sehr komplexen Deployment Descriptors. In Abbildung 2-4 ist ein einfaches Beispiel für eine Annotation abgebildet. Sie definiert für die dargestellte Klasse, dass es sich um eine Stateless Session Bean handelt.

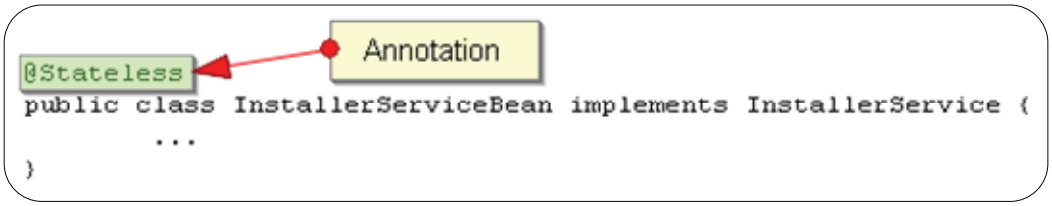


Abbildung 2-4: Beispiel einer Stateless Session Bean Annotation

2.8. Java Persistence API (JPA)

Die Java Persistence API dient der einfachen Abbildung von Java Objekten auf relationale Datenbanken. Hierzu wird eine Zwischenschicht definiert, welche sich transparent um das Lesen und Schreiben aus, beziehungsweise in die zugrunde liegende Datenbank kümmert. Durch den Mechanismus von JPA kann der Entwickler mit normalen Java Objekten

arbeiten, ohne sich selber um das Abbilden von diesen auf eine Datenbank kümmern zu müssen. Zum Benutzen von solchen Abbildungen ist es wichtig, dass die Datenbank Ressourcen vorher durch den Applikationsserver bereitgestellt wurden. Dies geschieht über die JDBC Ressourcen des Applikationsservers. Die JPA besteht neben der eigentlichen API aus drei Komponenten, die im Folgenden kurz beschrieben werden.

2.8.1. Persistent Entities

Persistent Entities bilden meist eine Tabelle einer Datenbank ab, wobei jede Instanz einer Zeile entspricht. Durch die Verwendung von Persistent Entities muss ein Entwickler sich nicht mehr selber um SQL Code bemühen, welcher für das Speichern, Lesen oder Ändern von Daten in der Datenbank benötigt wird.

2.8.2. Objektrelationelle Metadaten

Objektrelationelle Metadaten drücken die Beziehungen zwischen einzelnen Tabellen aus. Diese können entweder im Deployment Descriptor, oder, wie mit EJB 3.0 bevorzugt, über Annotationen definiert werden. Über diese Metadaten leitet sich das eigentliche Datenbankschema ab. Ein Beispiel für das Mapping von Java mit Annotationen auf ein Datenbankschema wird in Abbildung 2-5 gezeigt.

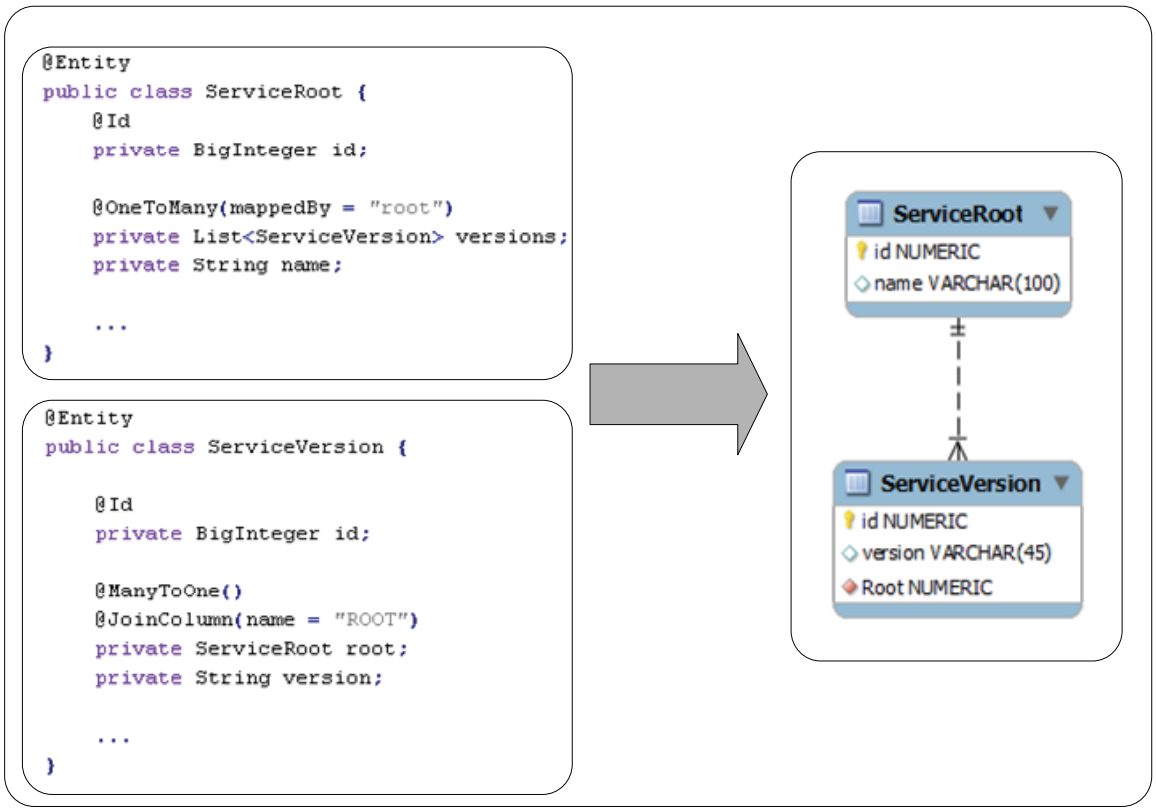


Abbildung 2-5: Mapping von Annotationen auf ein Datenbankschema

2.8.3. Java Persistence Query Language (JPQL)

Die Java Persistence Query Language dient der Abfrage von Entitäten aus der Datenbank. Sie ist stark an SQL angelehnt, bezieht sich jedoch auf Persistent Entities und nicht auf Datenbanktabellen. Abbildung 2-6 zeigt ein einfaches Beispiel einer solchen Abfrage, welche ein Objekt *ServiceRoot* mit dem Servicenamen „bankcheck“ abfragt.

```
SELECT r FROM ServiceRoot r
WHERE r.serviceName = 'bankcheck'
```

Abbildung 2-6: Beispiel einer JPQL Abfrage

2.9. Java Message Service (JMS)

Der Java Message Service dient zur Kommunikation zwischen Komponenten über Nachrichten (Messages). Hierbei müssen diese Komponenten nichts übereinander wissen, da sie lediglich Nachrichten an den Message Service schicken. Diese lose Koppelung wird häufig für das Einbinden von „Legacy[8]-Systemen“ verwendet. JMS stellt dabei zwei Verfahren zur Verfügung: eine Message Queue als Punkt-zu-Punkt Verbindung oder aber ein Topic als Anmelde-Versende-System. Wichtig ist, dass diese Queues oder Topics im Applikationsserver als Ressourcen angelegt werden, bevor diese von einem Service benutzt werden können.

2.10. Java Naming and Directory Interface (JNDI)

Mit dem Java Naming and Directory Interface können Daten und Objektreferenzen anhand eines Namens abgelegt werden. Dadurch können diese von Nutzern der Schnittstelle anhand ihrer spezifizierten Namen abgerufen werden. Unter JavaEE wird dies hauptsächlich dafür genutzt, EJBs oder Ressourcen an einen Namen zu binden und sie so leicht für Remote Zugriffe über diesen verfügbar zu machen.

2.11. Kontinuierliche Integration

Als Kontinuierliche Integration bezeichnet man einen Prozess aus der Softwareentwicklung, der ein ständiges „Neubuilden“ und Testen einer Anwendung beschreibt. Die Idee hinter der „Kontinuierlichen Integration“ ist, größere Codeänderungen in kleinen, inkrementellen Schritten durchzuführen, anstatt in einem Großen. Dieses

[8] Veraltete Technik oder Altlasten; Meist alte System die noch nicht abgelöst wurden und daher weiterhin genutzt werden.

Konzept erfordert jedoch von Seiten des Entwicklers eine regelmäßige Kontrolle der Versionsverwaltung. Den Namen erhält das Modell daher, dass kontinuierlich aus der Codebasis in der Versionsverwaltung ein Build erzeugt wird, sobald Änderungen auftreten. Ein Build bedeutet hierbei nichts anderes, als dass der Quellcode kompiliert wird und automatisierte Tests (zum Beispiel JUnit-Tests) ausgeführt werden. Laufen der Kompilier-Vorgang und die Tests ohne Fehler durch, so kann von einer stabilen Codebasis ausgegangen werden. Der Vorteil ist, dass durch das ständige Neukompilieren und -Testen Integrationsfehler, die bei der Arbeit von mehreren Entwicklern an einem Projekt auftreten können, sehr schnell zu finden sind. Zur Automatisierung dieses Vorgangs wird häufig ein Build System eingesetzt. Typische Vertreter sind hier Cruise Control, Hudson oder Anthill. In der vorliegenden Arbeit wurde als Build-System ein „Hudson Build-Server“ verwendet. Dieser dient als Repository, von dem alle zu installierenden Services bezogen werden.

3. Problemstellung

„Jede Lösung eines Problems ist ein neues Problem.“

(Johann Wolfgang von Goethe)

Im Folgenden wird exemplarisch auf die konkrete Problemstellung bei der QSC AG und deren Systemlandschaft eingegangen. Eine Anpassung an andere Umgebungen ist grundsätzlich möglich, wurde im Rahmen dieser Diplomarbeit jedoch nicht behandelt.

Um das Problem der Entwicklung von automatischen Installationsroutinen zu verstehen, ist es zunächst einmal notwendig, zu ergründen wie die Installation von Services in der vorliegenden Umgebung im Allgemeinen abläuft. Die Installation eines Service ist hierbei gleichbedeutend mit der Installation einer EJB in einen Applikationsserver. Genaugenommen wird eine EJB dem EJB-Container zugefügt. Da eine EJB aus normalen Java Klassen besteht, handelt es sich bei einem Service um ein von Java her bekanntes Jar-Archiv, welches die fertigkompilierten Java-Klassen des Services enthält. Dieses Jar-Archiv wird im konkreten Fall von einem „Hudson Build-Server“ erzeugt. Zur Installation eines Services wird zunächst das benötigte Jar-Archiv vom Build-Server geladen, konfiguriert und eventuell benötigte Ressourcen angelegt. Erst danach wird das Archiv über den Administrations-Bereich des Applikationsservers installiert.

In Abbildung 3-1 wird der gesamte Zyklus eines Service von der Entwicklung bis zur Installation bei der QSC AG schematisch dargestellt. Der „Deployer[9]“ hat hierbei drei Aufgaben zu bewältigen. Zunächst muss er sich den Build des Services herunterladen und konfigurieren. Erst danach kann er den Service deployen (installieren) und als letztes dokumentiert er seine Schritte in einem Wiki[10], um nachprüfbar zu dokumentieren, was er getan hat.

[9] Die Person die für die Installation von Services verantwortlich ist.
[10] Hawaianisch für „schnell“; Ein Hypertext-System für Webseiten, dessen Inhalte von Benutzern nicht nur gelesen, sondern auch direkt im Webbrowser geändert werden können.

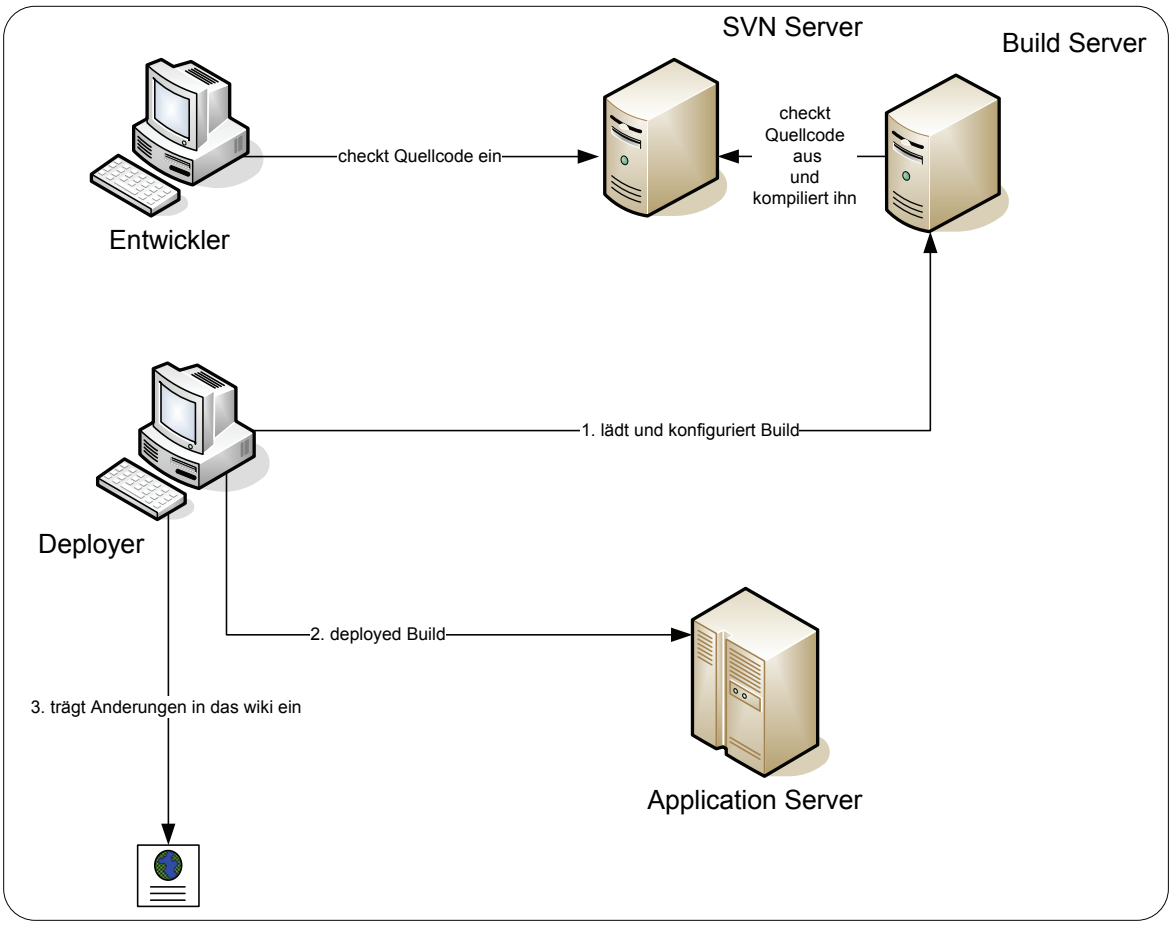


Abbildung 3-1: Von der Entwicklung zur Installation

Dieses Vorgehen ist für jeden zu installierenden Service zu wiederholen. Zu beachten ist auch, dass es mehr als einen Applikationsserver geben kann, auf dem installiert wird. So muss diese Prozedur nicht nur für jeden Service, der installiert werden soll, wiederholt werden, sondern eventuell auch für jeden Server. Bei diesem Vorgang müssen außerdem mögliche Abhängigkeiten zwischen Services und ihren unterschiedlichen Versionen beachtet werden. Bei einer größeren Server-Umgebung mit vielen Services, die aufeinander aufbauen, ist dies ein nicht-triviales Problem, bei dem der „Deployer“ schnell den Überblick verlieren kann. Das manuelle Vorgehen ist hierbei besonders fehleranfällig, da sich schnell Flüchtigkeitsfehler einschleichen können. Deswegen ist es das Ziel, möglichst viele dieser Schritte zu automatisieren oder zu vereinfachen. Im Folgenden werden noch einmal alle Schritte, die zum Installieren von einem Service nötig sind, betrachtet, sowie eventuelle Probleme hierbei aufgezeigt.

3.1. Abhängigkeiten erkennen

Durch das Zusammenfügen von Services kommt es zu Abhängigkeiten unter den einzelnen Komponenten. Diese entstehen zumeist aus der Situation, dass ein Service einer höheren Ebene sich aus mehreren Services der niedrigeren Ebenen zusammensetzen kann. Diese Abhängigkeiten müssen beim Installieren erkannt und aufgelöst werden, da ein Service ohne seine Abhängigkeiten nicht funktioniert. Im Moment werden solche Abhängigkeiten manuell gepflegt, indem sie in der Dokumentation eines Services vermerkt werden.

Diese Abhängigkeiten müssen bei der manuellen Installation zunächst gesichtet werden, um aus ihnen einen Abhängigkeitsbaum zu erstellen. Nicht selten muss ein „Deployer“ daher nicht nur einen Service, sondern gleich mehrere installieren, wenn Abhängigkeiten vorliegen, die sich auf dem Applikationsserver noch nicht in der richtigen Version befinden. Die Dokumentation der Services, und somit auch deren Abhängigkeiten, werden hierbei zentral in einem Wiki bereitgestellt, wo sie von Hand gepflegt werden. Ziel ist es jedoch, solche Abhängigkeiten automatisch zu erkennen, zu speichern und in einer wieder verwendbaren Form zentral zur Verfügung zu stellen.

3.2. Laden von Services vom Build-Server

Die fertig kompilierten Jar-Archive der Services liegen auf einem Build-Server und müssen von diesem geladen werden. Er dient dabei als Repository für fertig kompilierte Services. Zu installierende Services werden ausschließlich von diesem Server bezogen. Derzeit muss per Hand jedes einzelne Archiv heruntergeladen werden (siehe Abbildung 3-2), um es später auf dem Server zu installieren.

Ziel ist es, automatisiert alle Services und deren Versionen auf dem Build-Server zu erkennen, um bei einer Installation den konkret benötigten Build automatisch herunterzuladen und anschließend weiterzuverarbeiten.

Abbildung 3-2: Übersichtsseite eine Projektes vom Build-Server

3.3. Konfiguration von Services

Zu installierende Services müssen vor ihrer Installation konfiguriert werden. Dies beinhaltet unter anderem das Anpassen des Deployment Descriptors sowie eventuell vorhandene andere Konfigurationsdateien. Außerdem kann ein Service eine oder mehrere Ressourcen benötigen. Diese Ressourcen müssen bei Bedarf erst im Applikationsserver angelegt und konfiguriert werden, damit sie vom Service genutzt werden können. In Abbildung 3-3 wird das Administrationsinterface zum Anlegen von JDBC Ressourcen dargestellt.

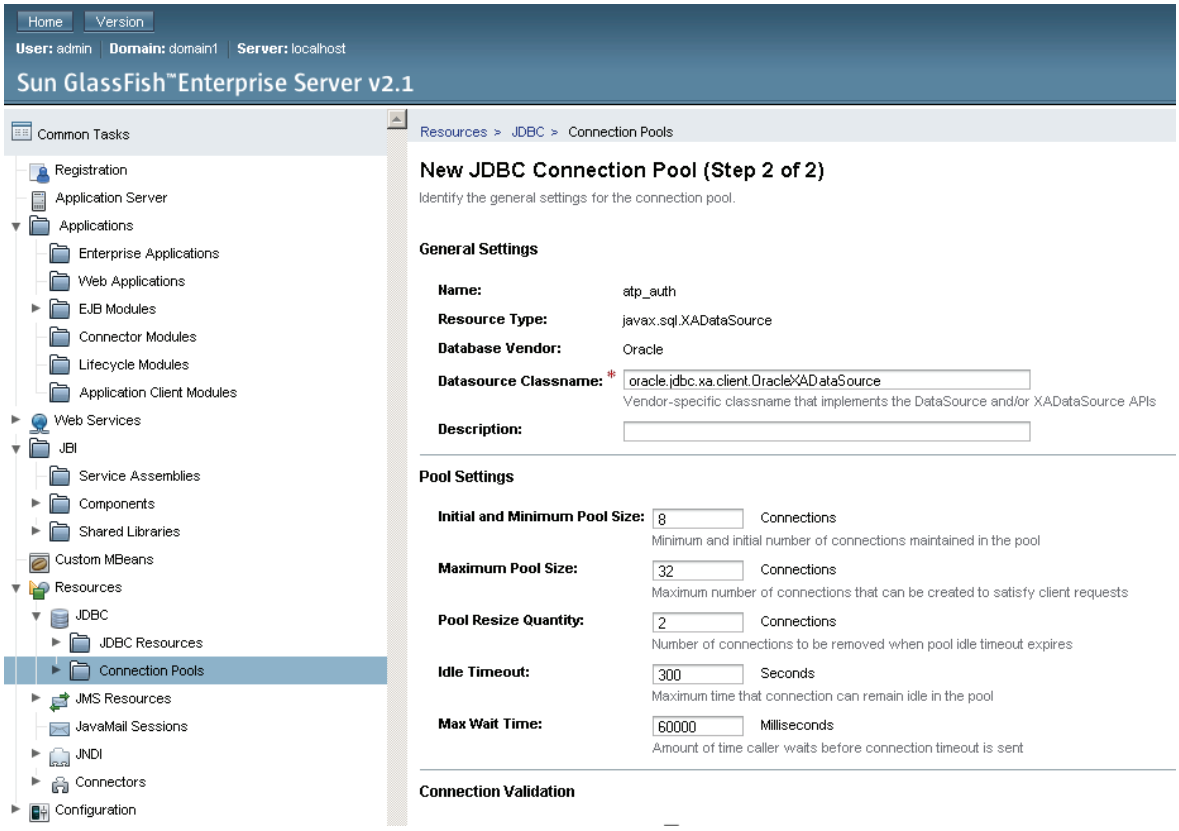


Abbildung 3-3: Administrationsinterface zum anlegen von JDBC Ressourcen auf einem Glassfish Applikationsserver

Aktuell wird diese Konfigurationsarbeit manuell durchgeführt. Konkret bedeutet dies, dass zur Konfiguration jeden Services zunächst alle benötigten Ressourcen über das Web-Interface angelegt werden müssen. Danach ist das Jar-Archiv zu entpacken, um eventuelle Konfigurationsdateien zu bearbeiten. Nach deren Bearbeitung müssen sie wieder zu einem Jar-Archiv gepackt werden, um dieses auf dem Applikationsserver installieren zu können. Im Zuge der Entwicklung von automatischen Installationsroutinen soll – wenn möglich – sowohl die Konfiguration, als auch das Anlegen von Ressourcen dem Nutzer abgenommen, oder zumindest eine einfache Möglichkeit zur Automatisierung dieser Prozesse entwickelt werden.

3.4. Installieren / Deployen von Services

Das eigentliche Installieren, Deployment genannt, muss explizit angestoßen werden. Dies bedeutet das Verfügbarmachen des vorher konfigurierten Pakets. Dazu wird es auf den Server hochgeladen und danach vom EJB-Container des Servers verwaltet, so dass es für die Benutzung zur Verfügung steht. Momentan wird dies über das Administrations Frontend des Applikationsservers vorgenommen (siehe Abbildung 3-4).

Dieses Vorgehen ist für eine große Zahl von Services jedoch nicht sehr praktikabel, da jeder Service einzeln über das Interface ausgewählt und installiert werden muss. Ziel ist es also, mit einfachen Methoden einen Service inklusive seiner Abhängigkeiten installieren zu können. Dies soll mit nur wenigen Schritten ermöglicht werden.

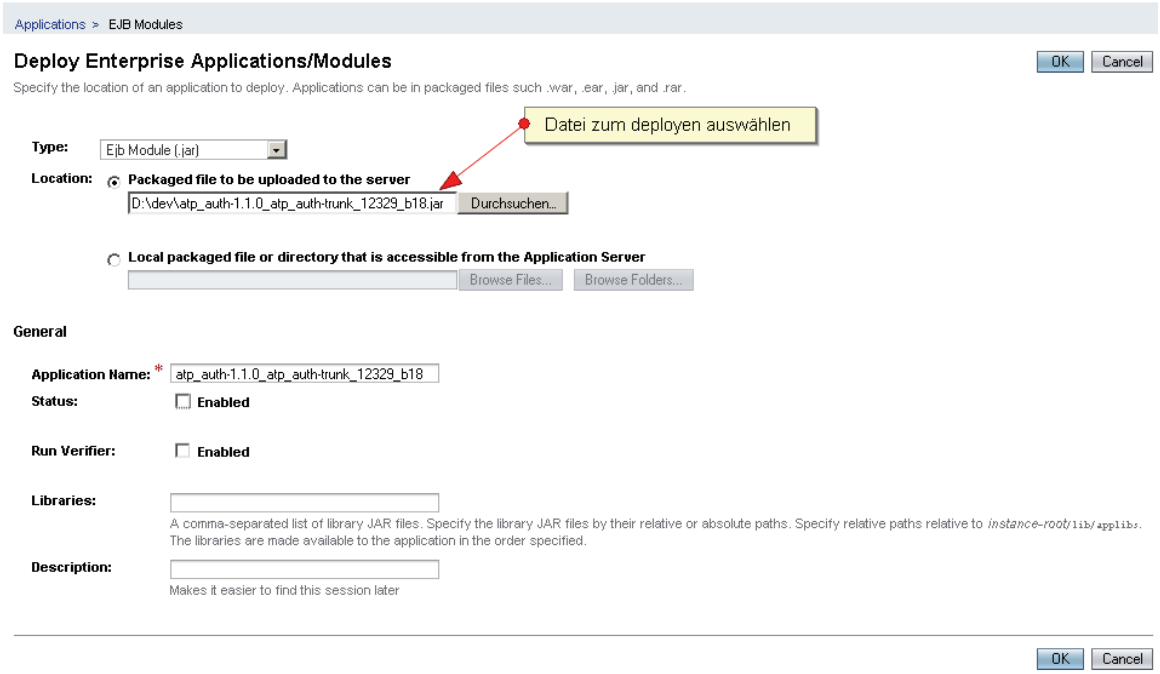
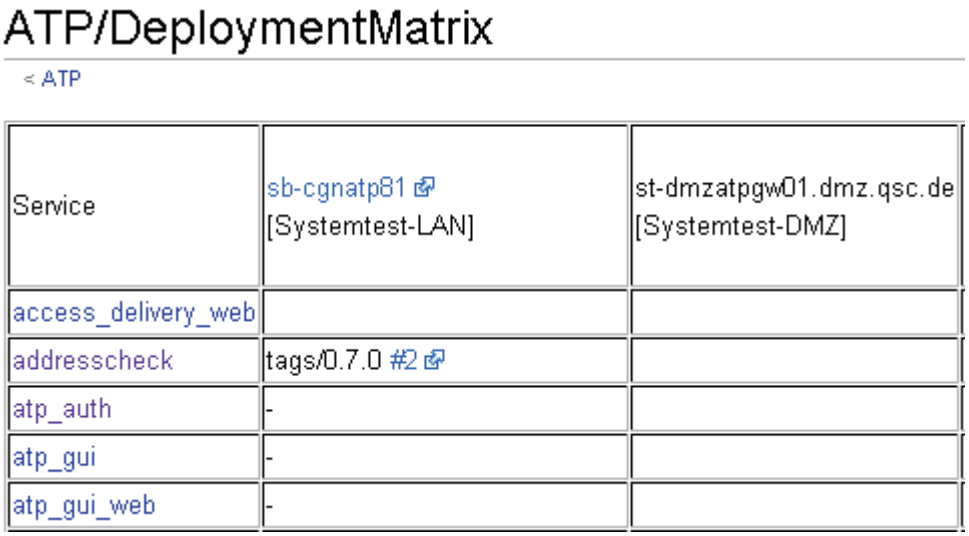


Abbildung 3-4: Administrations Frontend zum Deployen eines Glassfish Applikationsserver

3.5. Anlegen einer History

Nach dem Installieren wird in einem Wiki in der Deployment-Matrix festgehalten, welcher Service gerade installiert wurde und auf welchem Server dies geschehen ist. Abbildung 3-5 zeigt ein Beispiel für eine solche Matrix in der Wiki. Durch die History wird der letzte Zustand eines Servers und seiner Services festgehalten. Nachteil an dem Wiki-Verfahren ist zum Einen, dass alle Änderungen manuell gepflegt werden müssen, und zum Anderen, dass nur der letzte Installationsstand auf einem Server zu sehen ist. Wünschenswert wäre eine lückenlose History, um auch ältere Stände der installierten Services betrachten zu können. Somit könnten auch alte Konfigurationen wiederhergestellt werden.



ATP/DeploymentMatrix

< ATP

Service	sb-cgnatp81 [Systemtest-LAN]	st-dmzatpgw01.dmz.qsc.de [Systemtest-DMZ]
access_delivery_web		
addresscheck	tags/0.7.0 #2	
atp_auth	-	
atp_gui	-	
atp_gui_web	-	

Abbildung 3-5: Beispiel einer Deployment-Matrix

4. Lösungsansätze

Für die Lösung der vorher beschriebenen Probleme gibt es mehrere Ansätze, die im Folgenden kurz vorgestellt werden.

4.1. Fertige Produkte

Die Problematik der automatischen Installation von Services auf einem Applikationsserver ist grundsätzlich nicht neu. Deswegen verwundert es kaum, dass es bereits mehrere fertige Software Lösungen zu diesem Problem gibt. Einige der bereits verfügbaren Produkte werden nachfolgend kurz vorgestellt.

4.1.1. asadmin

Zu den Applikationsservern werden häufig konsolenbasierte Werkzeuge zur Verwaltung und Installation mitgeliefert. Bei dem Glassfish-Server, der in dieser Arbeit verwendet wurde, heißt dieses Werkzeug `asadmin`, mit dem einfache Aufgaben der Weboberfläche ersetzt werden können. So ist es möglich von der Konsole aus ein Jar-Archiv zu installieren oder Ressourcen anzulegen. Ein vollautomatischer Ablauf ist hier jedoch nicht vorgesehen. Dieser müsste durch Skripte realisiert werden, welche die Aufgaben nacheinander abarbeiten. Einer der größten Nachteile ist jedoch, dass es keine Auflösung der Abhängigkeiten gibt. So wäre es möglich, skriptgesteuert einzelne Services mit ihren Ressourcen zu installieren, jedoch würden eventuelle Abhängigkeiten nicht automatisch mitinstalliert. Eine Funktion zur Historisierung von Installationen ist ebenso wenig vorgesehen.

4.1.2. Deployit

Ein anderes Produkt, welches zur automatischen Installation von Services eingesetzt werden kann, ist Deployit der Firma Xebia Labs[11]. Mit Deployit wird zunächst über so genannte CIs (Configuration Items) die Systemlandschaft abgebildet. Das heißt, jedes CI stellt einen Teil der Infrastruktur dar. Dies geht von einem CI, welches einen Host darstellt, bis zu CIs, welche Applikationsserver auf einem Host darstellen. Außerdem werden Applikationen (Services) auch durch CIs dargestellt. Die Idee hinter Deployit ist, dass zunächst der erwünschte neue Stand der Systemlandschaft modelliert wird, um dann die

[11] http://www.xebialabs.com/

notwendigen Schritte zu berechnen, um den gewünschten Endzustand zu erreichen. In Abbildung 4-1 ist das Webinterface von Deployit zum Modellieren solcher CIs zu sehen.

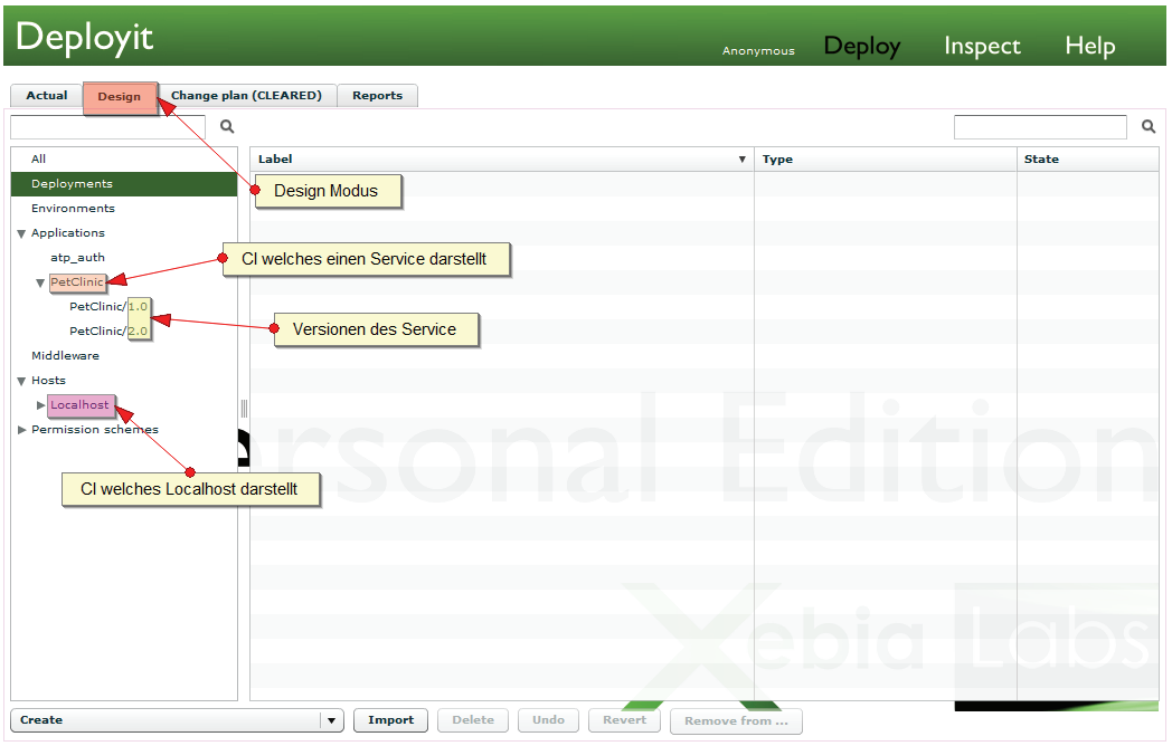


Abbildung 4-1: Design Interface des Tools Deployit

Aus den aus der Modellierung abgeleiteten Schritten wird ein Ausführungsplan erstellt, welcher dann vom Benutzer angestoßen werden kann, um die Änderungen auszuführen. Abbildung 4-2 zeigt einen erstellten Ausführungsplan.

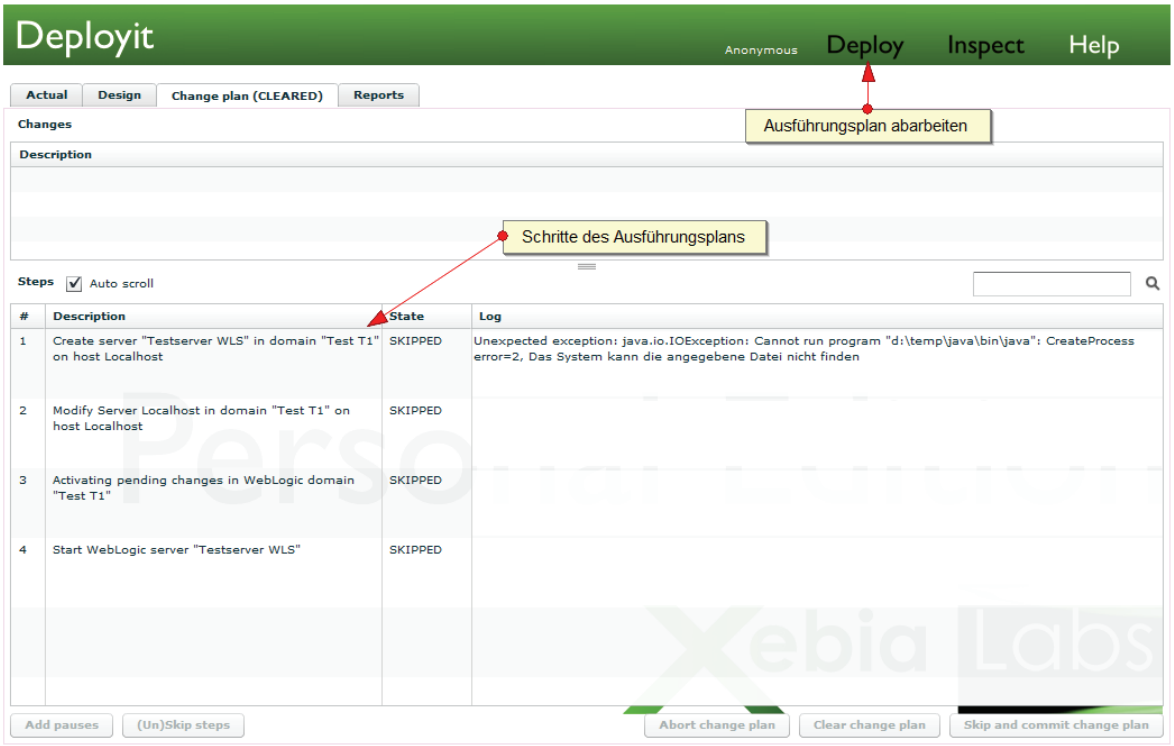


Abbildung 4-2: Ausführungsplan des Tools Deployit

Soll ein Service installiert werden, so wird dieser zunächst auf dem entsprechenden Server modelliert und durch das Anstoßen des Ausführungsplans letztendlich installiert. Deployit unterstützt außerdem eine Historisierung durch die sogenannten *Reports*. Mit diesen ist es möglich, Ausführungspläne noch einmal zu betrachten oder alle Deployments eines Servers nachzuvollziehen.

Die Stärke von Deployit ist die Darstellung der gesamten Systemlandschaft und ihrem Zustand. Dies ermöglicht das schnelle Installieren auf verschiedenen Umgebungen.

Der Nachteil von Deployit ist die fehlende Unterstützung von externen Repositorys. Alle Services müssen erst in Deployit als CIs verfügbar gemacht werden. Auch eine Abhängigkeitsauflösung von Services ist nicht vorgesehen. Ein weiterer Nachteil in Bezug auf die für diese Arbeit vorliegende Aufgabenstellung ist, dass der verwendete Glassfish Applikationsserver nicht unterstützt wird.

4.1.3. AutoDeploy

AutoDeploy aus der BuildProcess Toolbox[12] ist ein weiteres Tool, welches sich der automatischen Installation von Services widmet. AutoDeploy setzt sich hierbei aus zwei Komponenten zusammen: Zum Einen den *Agents,* welche sich direkt auf der Hardware des Applikationsservers befinden müssen, um die eigentliche Installation durchzuführen, zum Anderen dem Web-Frontend, über den alle Agents verwaltet werden.

Installationsparameter werden bei AutoDeploy durch eine XML-Datei beschrieben. In dieser XML-Datei wird definiert, auf welchen Servern installiert werden soll und welche Ressourcen dort angelegt werden müssen.

Ein Vorteil von AutoDeploy ist die Möglichkeit der automatischen Überwachung eines Build-Servers auf neue Versionen. Hierdurch ist es möglich, ein Update sofort zu installieren, sobald eine neue Version zur Verfügung steht. Durch die flexible Konfiguration ist sowohl das Erstellen von Ressourcen möglich, sowie das Ersetzen von Konfigurationsdateien. Auch AutoDeploy bietet keinerlei Möglichkeit, Abhängigkeiten aufzulösen. Eine Historisierung ist ebenso wenig vorgesehen. Der für die Arbeit verwendete Glassfish Applikationsserver zählt ebenso wie bei Deployit nicht zu den unterstützten JavaEE Plattformen.

4.2. Anpassung von Software mit ähnlichen Funktionen

Ein anderer Ansatz wäre die Anpassung eines bereits verfügbaren Installationswerkzeuges an die gestellten Anforderungen. Vorstellbar wäre hier eine Anpassung von Deployit oder AutoDeploy. Während Deployit eine Plugin Schnittstelle zur Verfügung stellt, stehen die Quelltexte von AutoDeploy unter der Apache License 2.0[13] zur Verfügung. Problematisch wäre hier, dass tiefere Eingriffe in die Systeme nötig wären, um bei beiden Anwendungen zunächst die Glassfish Applikationsserver Unterstützung zu implementieren.

Eine andere Möglichkeit wäre beispielsweise, einen von Linux bekannten Paketmanager wie `aptitude` oder `rpm` anzupassen. Vom Prinzip her stellen Paketmanager genau die Funktionalität zur Verfügung, die für eine Automatisierung des Installationsprozesses benötigt würde. Sie installieren Pakete und lösen dessen Abhängigkeiten auf, arbeiten jedoch auf einer anderen Ebene der Software Installation. Während JavaEE Services in eine Middleware, nämlich den Applikationsserver, installiert werden, handelt es sich bei

[12] http://buildprocess.sourceforge.net
[13] http://www.opensource.org/licenses/apache2.0.php

den von den Paketmanagern verwalteten Paketen um Programme, welche direkt in das Betriebssystem integriert werden müssen. Allein dieser Umstand erfordert eine komplexe Anpassung, da die Anwendungen für dieses Einsatzszenario nicht gedacht sind. Des Weiteren unterstützen sie von Haus aus nur einen einfachen Support für eine History. Meist kennen sie nur den aktuellen Zustand des Systems.

Da jedoch eine umfangreiche Historisierung eine der Haupt-Anforderungen für den Einsatzzweck ist, wären auch hier größere Anpassungen vorzunehmen.

4.3. Eigenentwicklung

Da keine der bis jetzt vorgestellten Lösungen allen Anforderungen gerecht wird, liegt die Überlegung nahe, ein eigenes Konzept zu entwickeln. Dieses kann genau auf die Problemstellung zugeschnitten werden und erfüllt idealerweise alle Anforderungen. Zudem wären spätere Erweiterungen oder Anpassungen des Systems einfacher möglich, da die komplette Codebasis und Architektur in einer Hand liegen.

4.4. Gewählte Lösung

In Tabelle 4-1 werden noch einmal alle Vor- und Nachteile der einzelnen Produkte aufgezeigt.

Produkt	**Vorteile**	**Nachteile**
asadmin	+ Anlegen von Ressourcen	– Keine Auflösung von Abhängigkeiten – Keine Historisierung – Keine eigene Automatisierung
Deployit	+ Gute Übersicht über Systemlandschaft + Umfangreiche Historisierung	– Keine Auflösung von Abhängigkeiten – Keine Unterstützung des Glassfish Applikationservers – Kein erstellen von Ressourcen – Schwierige Anpassung
AutoDeploy	+ Flexible Konfiguration + Anlegen von Ressourcen	– Keine Auflösung von Abhängigkeiten – Keine Unterstützung des Glassfish Applikationsservers – Keine Historisierung
Eigenentwicklung	+ Auf Anforderungen zugeschnitten	– Eventuell längere Entwicklungstzeit

Tabelle 4-1: Vor und Nachteile von verschiedenen Deployment Produkten

Nach Abwägung der Vor- und Nachteile wurde das Konzept einer Eigenentwicklung gewählt.

Dies hat mehrere Gründe. Zum Einen soll sich das Endprodukt so gut wie möglich in das bestehende Umfeld einfügen, ohne große Änderungen an den vorhandenen Systemen vornehmen zu müssen. Zum Anderen erscheint eine Eigenentwicklung in Hinblick auf die Wartbarkeit sinnvoll. So könnte es zwar kurzfristig geeignet erscheinen, bestehende Produkte so anzupassen, dass sie den Bedürfnissen gerecht werden, jedoch könnte kein Einfluss auf die Entwicklung dieser „Fremd“-Produkte genommen werden. So könnten beispielsweise größere Änderungen in der Basis des verwendeten Produktes eigene Modifikationen so beeinträchtigen, dass Anpassungen ständig überarbeitet werden müssten, oder – im schlimmsten Fall – gar nicht mehr funktionieren.

5. Verwendete Produkte

5.1. Entwicklungsumgebung

5.1.1. Eclipse

Zu der Entwicklung des Javacodes für den Installations-Service, sowie für den Kommandozeilen-Klienten, wurde ein von der QSC für die ATP-Plattform[14] angepasstes Eclipse in der Version 3.5 eingesetzt. Die Anpassungen ermöglichen ein einfacheres Entwickeln von Services auf Basis von ATP. Durch das Erstellen von UML-Diagrammen wird ein Großteil des Javacodes durch Generatoren, die in Eclipse integriert wurden, erzeugt. Dieser modellgetriebene Ansatz ermöglicht Kürzere Entwicklungszeiten. Zusätzlich dienen die UML-Diagramme als Dokumentation zu den erstellten Software Projekten.

5.1.2. Magicdraw

Magicdraw ist ein Programm der Firma „No Magic“ zum Modellieren von UML-Diagrammen jeglicher Ausprägung. Bei der Entwicklung im ATP Umfeld dient es zur Umsetzung der modelgetriebenen Entwicklung und somit zur Erstellung eines Modells in Form eines UML-Klassendiagramms.

Das Modell ist hierbei eine Abstraktion, die zunächst von der Programmiersprache unabhängig ist und erst durch entsprechende Generatoren in eine oder mehrere konkrete Programmiersprachen umgesetzt wird. Das so generierte Codegerüst muss jetzt nur noch mit konkretem Befehlen gefüllt werden.

Alle in dieser Arbeit abgebildeten Klassendiagramme wurden mit *Magicdraw* erstellt.

5.1.3. Versionsverwaltung

Zur Versionsverwaltung wird bei der QSC AG das Versionsverwaltungssystem *Subversion* eingesetzt. Mit diesem ist es möglich, größere Projekte mit mehreren Entwicklern zu koordinieren. Durch die zentrale Ablage jedes erzeugten Quellcodes ist es möglich, den Entwicklungsstand jedes Projektes nachzuvollziehen oder frühere Entwicklungsstände wiederherzustellen.

[14] Siehe 5.2

Zur einfachen Anbindung an das Subversion-System wurde der in Eclipse leicht zu integrierende Client „Subclipse“ benutzt.

5.2. Advanced Tool Plattform (ATP)

Die Advanced Tool Plattform ist eine von der QSC AG selber entwickelte SOA[15]-Plattform zur einheitlichen Entwicklung und Betrieb von Services auf der Basis von JavaEE. Hierbei stellt die Plattform eine Grundbasis zur Entwicklung zur Verfügung. So wird auf Basis von ATP der modellgetriebene Entwicklungsansatz in den Vordergrund gestellt. Das heißt, es wird zunächst ein Modell in UML erstellt, welches später durch Generatoren in Java-Code umgewandelt wird. Die Basis zum Betrieb stellt ein „Glassfish Enterprise Server“ im derzeitigen Stand 2.1 zur Verfügung. Außerdem enthält die Plattform zahlreiche Java Bibliotheken, welche als Teil der Plattform in der Entwicklung von neuen Services genutzt werden können. Ziel der Plattformumgebung ist es, einen wohldefinierten Zustand sowohl zur Entwicklung als auch zum Betrieb zur Verfügung zu stellen.

Der entwickelte Prototyp wurde ebenfalls mit Hilfe der ATP-Umgebung umgesetzt.

5.3. JavaEE

5.3.1. Java Specification Request (JSR) 88

Der Java Specification Request wurde definiert, um neue Java-Standards, oder Erweiterungen für die Java-Programmiersprache, oder deren Laufzeitumgebung zu entwickeln. Hierzu werden Anforderungen (Requests) geschrieben, welche an das von Sun Microsystems betriebene „Process Managment Office“ gestellt werden. Ziel soll es sein, gemeinschaftlich mit der Java-Community die Weiterentwicklung von Java voranzutreiben. Für jede vorgeschlagene Änderung wird ein eigener JSR erstellt, welcher fortlaufend nummeriert wird. Viele der bereits in Java eingeführten Erweiterungen sind aus einem JSR entstanden.

In dem JSR88 wird eine einheitliche Schnittstelle zum Deployen von JaveEE Anwendungen auf einen JavaEE Applikationsserver beschrieben. Diese Spezifikation wurde bereits verabschiedet und wird normalerweise von jedem gängigen JavaEE Applikationsserver umgesetzt. Auf der Grundlage dieser Spezifikation ist es möglich, eine

[15]Service Oriented Architecture; ein Architekturmuster aus der Informationstechnik, das Services zur Kapselung von Geschäftsprozessen nutzt.

Anwendung für das Deployment zu entwickeln, welches von dem konkret verwendeten Applikationsserver unabhängig ist.

Der im Rahmen dieser Arbeit entwickelte Prototyp verwendet diese Spezifikation für das Deployment der Services. Dies ermöglicht dem Prototypen grundsätzlich auch mit anderen Applikationsservern zusammenzuarbeiten, außer dem verwendeten Glassfish Enterprise Server.

5.3.2. Java Management Extension (JMX)

JMX ist eine vom Java Community Process entwickelte Erweiterung zur Überwachung und Verwaltung von Java-Anwendungen. Sie ermöglicht den Zustand von Komponenten über so genannte *Managed Beans* (MBeans) zur Laufzeit auszulesen oder zu verändern. JMX ist dazu gedacht eine einheitliche Schnittstelle zum Verwalten von Anwendungen und Komponenten zur Verfügung zu stellen. Häufig findet JMX Verwendung in der Verwaltung eines JavaEE Applikationsserver. In der vorliegenden Arbeit wird JMX als Schnittstelle zum Applikationsserver benutzt, um Ressourcen im Applikationsserver anzulegen.

5.3.3. Java API for XML Binding (JAXB)

JAXB steht für "Java API for XML Binding". JAXB definiert eine Methode um Java Objekte in XML zu überführen und umgekehrt. Möglich wird dies durch die in Java 5 eingeführten Annotationen. Diese ermöglichen es, Metadaten innerhalb eines Java Quelltextes zu definieren. So reichen zum Beispiel die in Abbildung 5-1 gezeigten Annotationen, um im Quellcode eines Java Objektes zu definieren, wie das aus dem Objekt generierte XML aussehen soll.

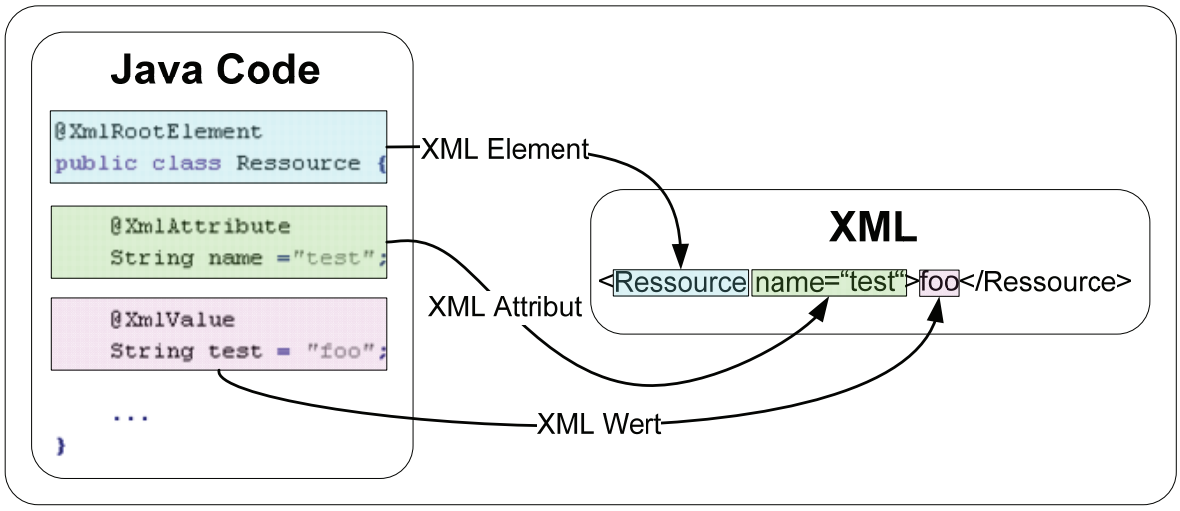


Abbildung 5-1: Beispiel zur Umwandlung von Java Objekten nach XML durch JAXB

Es ist mit JAXB jedoch ebenso möglich, aus einem XML-Schema diese bereits annotierten Java Objekte generieren zu lassen oder aus annotierten Objekten ein XML-Schema zu erzeugen. Durch diese einfachen, jedoch sehr mächtigen Werkzeuge ist es mit wenigen Zeilen Quellcode möglich, annotierte Objekte in XML zu überführen. Ein Beispiel hierfür wird in Abbildung 5-2 gezeigt.

```
JAXBContext context = JAXBContext.newInstance(Ressource.class);
Marshaller marshaller = context.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(new Ressource(), System.out);
```

Abbildung 5-2: Beispielcode zum Umwandeln von Java Objekten mittels JAXB

In der Abbildung ist zu sehen, dass nur wenige Schritte zum Umwandeln eines Java Objektes in XML nötig sind. Zunächst wird ein `JAXBContext` erstellt, dem die annotierte umzuwandelnde Klasse übergeben wird. Aus diesem lässt sich nun ein so genannter `Marshaller` erzeugen, welcher die eigentliche Umformung übernimmt. Wird auf diesem `Marshaller` nun die Funktion `marshal` mit einer konkreten Instanz des Objektes aufgerufen, so wird dieses in XML umgewandelt. Der umgekehrte Weg ist ebenso einfach über einen `Unmarshaller` zu realisieren.

5.4. Hudson-Build Server

Zur Umsetzung der kontinuierlichen Integration kam ein Hudson Server in der Version 1.318 zum Einsatz. Die Aufgabe des „Hudson“ besteht darin, den Quellcode zu kompilieren und automatisierte Tests auf diesem auszuführen. Außerdem stellt er nach einem erfolgreichen Build diesen als Jar-Archiv zur Verfügung. Im Rahmen dieser Arbeit diente der „Hudson“ als Repository, aus dem die zu installierenden Services geladen wurden.

5.5. Glassfish Enterprise Server

Der „Glassfish Enterprise Server“ ist eine Open Source Implementierung der JavaEE Spezifikation, welche von der Firma Sun Microsystems betreut wird. Er dient in der vorliegenden Arbeit als Laufzeitumgebung für alle Services. Eingesetzt wurde die der JavaEE Spezifikation 5 entsprechende Version 2.1 des Applikationsservers.

6. Implementierung

Die Umsetzung der automatischen Installationsroutinen erfolgt seinerseits als Service. Dies hat mehrere Vorteile. Zum Einen wird die eigene Funktionalität gekapselt. Somit ist es anderen Services möglich, den Installationsservice zu nutzen, um wiederum neue kombinierte Funktionalitäten bereitzustellen. Zum Anderen wird durch den Aufbau als Service eine zentrale Anlaufstelle zur Verwaltung von Services über die gesamte Serverlandschaft bereitgestellt. Vor allem die Historisierung, die durch den Service bereitgestellt wird, ist so auch einfach für andere Zwecke nutzbar. Zum Beispiel ist dadurch die Erstellung einer dynamischen Webseite denkbar, welche die verschiedenen Zustände zu unterschiedlichen Zeitpunkten visualisiert und damit die Übersichtsseite im Wiki ersetzt.

Der Nachteil, der durch die Umsetzung als Service entsteht, ist, dass die Installation selbst eine Laufzeitumgebung in Form eines Applikationsservers benötigt. Dies bereitet jedoch keine Probleme, da alle anderen zu installierenden Services ebenso einen Applikationsserver benötigen, dieser also ohnehin schon vorhanden sein muss.

6.1. Katalogisieren der Builds

Um einen Überblick über die installierbaren Services zu bekommen, ist es als erstes notwendig, die Builds aus dem Hudson-Server zu katalogisieren. Hierfür wurde ein SAX[16]-Parser geschrieben, welcher über die XML-API des „Hudson“ die Informationen zu allen verfügbaren Services sammelt und diese in einer eigenen Datenstruktur speichert. So wird ein Abbild aller im Build-Server verfügbaren Services und deren Versionen in der Datenbank angelegt.

In Abbildung 6-1 ist ein Ausschnitt der Ausgabe der XML-API des Hudson-Server dargestellt. Hierbei handelt es sich um ein einfaches XML Dokument, welches alle benötigten Informationen zu einem bestehenden Service und dessen Builds zusammenfasst.

[16] Abkürzung für Simple Api for XML. SAX definiert eine API zum sequentiellen parsen von XML Dateien.

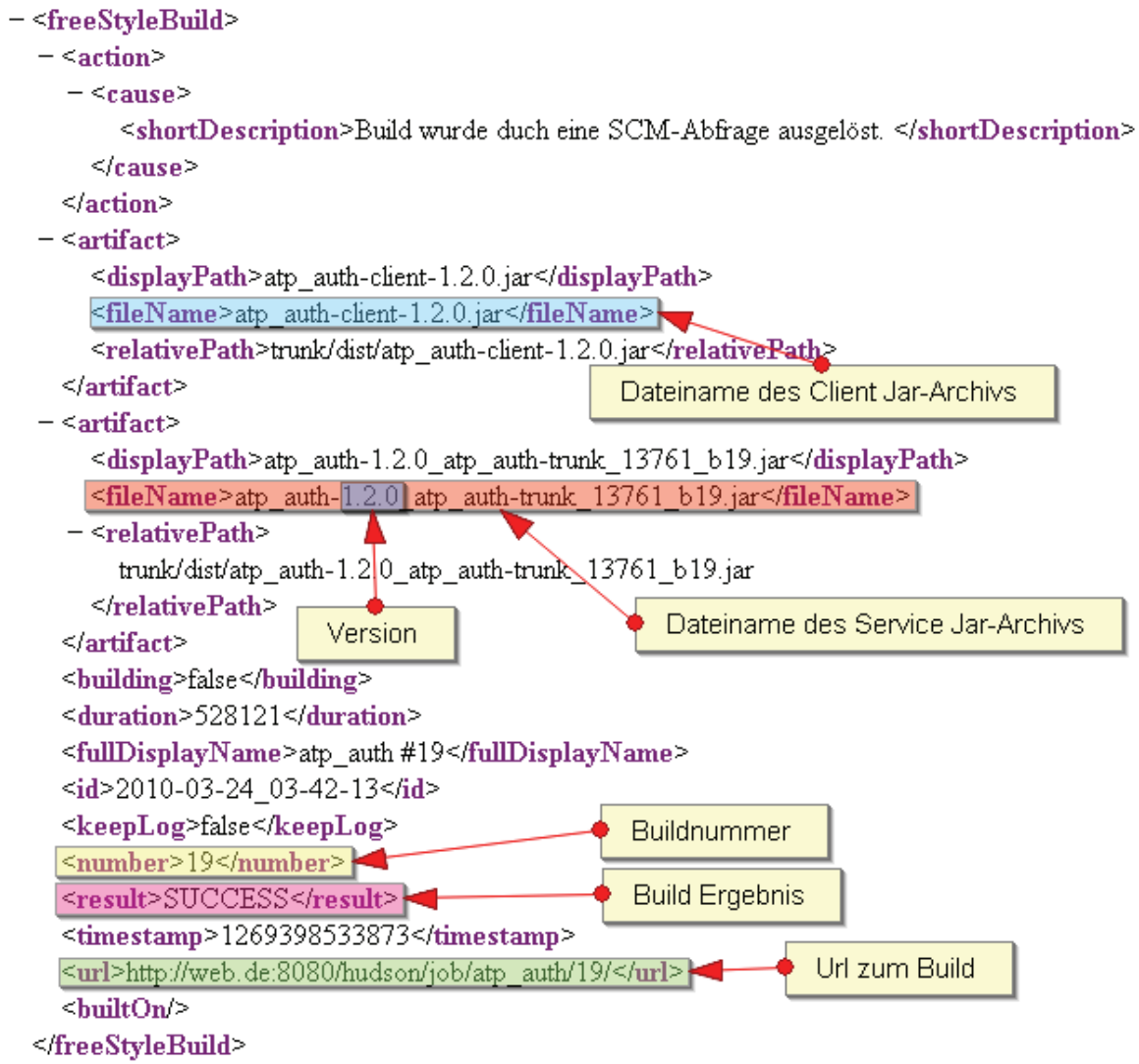


Abbildung 6-1: Ausgabe der XML-API eines Hudson Build-Servers

Die für die Installation und Abhängigkeitsauflösung relevanten Informationen werden hierbei in eine eigene Datenstruktur überführt.

Diese Datenstruktur ist so gewählt, dass sie alle benötigten Informationen erfasst. Hierzu zählen der Name des Service, seine Beschreibung, sowie die URL, unter der dieser im Hudson-Server zu finden ist. Zur Optimierung des Parser-Vorgangs wird außerdem die Nummer des letzten Build-Prozesses gespeichert. Bereits abgeschlossene Build-Vorgänge ändern sich nicht mehr und müssen somit nicht beim erneuten Parsen berücksichtigt werden. Zu jedem so erfassten Service wird jede vorhandene Version (Build) gespeichert. Hierbei werden nur erfolgreiche Builds in die Datenstruktur übernommen, da davon ausgegangen wird, dass nur funktionierende Services installiert werden sollen. Der Aufbau einer separaten Datenstruktur in einer Datenbank für die vom Build-Server bereitgestellten Services hat mehrere Gründe. Zum Einen ist das Parsen der XML-Dateien des Build-

Servers relativ langsam im Vergleich zu Abfragen, die an eine Datenbank gestellt werden. Zum Anderen muss ohnehin eine Struktur erstellt werden, welche die Abhängigkeiten zwischen den einzelnen Services abbildet. In Abbildung 6-2 wird die gewählte Datenstruktur in einem Klassendiagramm dargestellt.

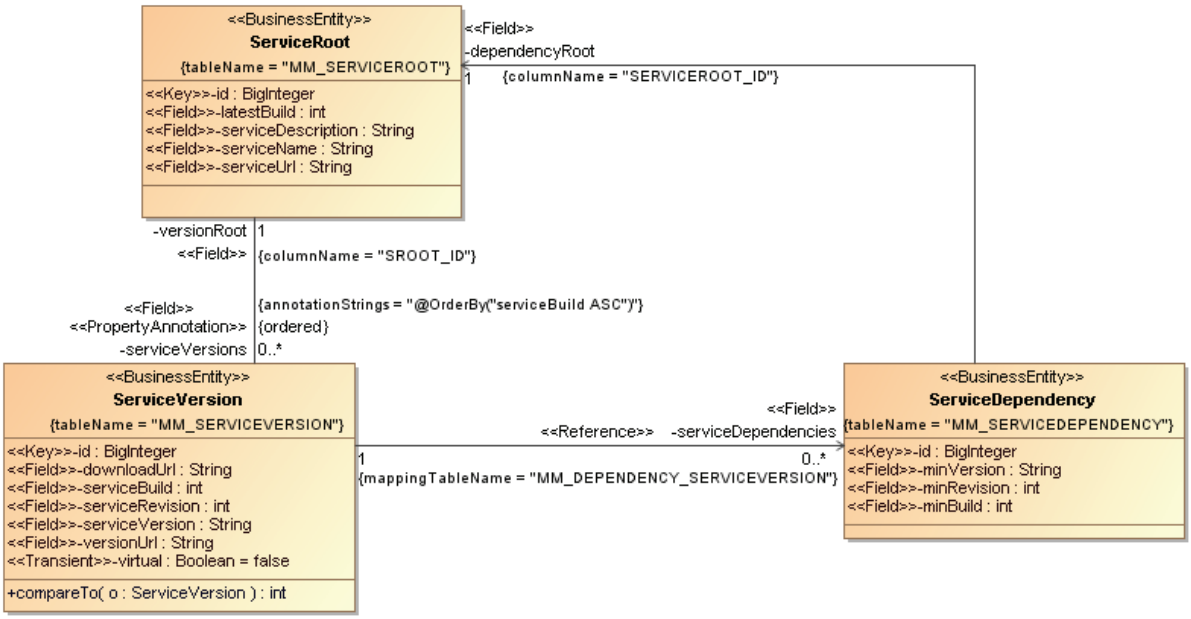


Abbildung 6-2: Klassendiagramm der Services, Versionen und Abhängigkeiten

Wie aus dem Klassendiagramm ersichtlich, kann jeder Service (ServiceRoot) mehrere Versionen haben. Jede Version kann wiederum mehrere Abhängigkeiten definieren. Hierdurch wird eine Grundlage geschaffen, um auch komplexe Abhängigkeiten abbilden zu können.

6.2. Abhängigkeiten erkennen

Das Erkennen von Abhängigkeiten ist recht komplex. Grundsätzlich ist von Außen nicht ersichtlich, welche anderen Services ein Service aufruft. Dafür müsste eine Analyse des Quellcodes durchgeführt werden, was nicht praktikabel ist, da es sehr lange dauern würde, wenn ein Service aus sehr vielen Klassen besteht und jede einzelne Klasse analysiert werden müsste. Der wichtigste Grund, der gegen eine Quellcodeanalyse spricht, ist jedoch, dass zwar festgestellt werden könnte, dass ein anderer Service aufgerufen wird, und meist auch noch wie dieser heißt, jedoch nicht um welche Version es sich handelt. Somit würde eine Quellcodeanalyse zwar Abhängigkeiten zu anderen Services aufzeigen, jedoch eine Zuordnung zu bestimmten Versionen dennoch nicht ermöglichen.

Ein einfacherer Weg führt über die Datei *.classpath*, die ebenfalls im temporären Verzeichnis des Build-Servers gespeichert ist. Diese wird von der Entwicklungsumgebung Eclipse angelegt und ist in jedem mit Eclipse erstellten Projekt vorhanden, im konkreten Fall also für jeden Service. In dieser XML-Datei werden unter anderem alle Bibliotheken aufgeführt, die in einem Eclipse Projekt eingebunden worden sind. Da gemäß der ATP-Konventionen eine EJB immer ein Business-Interface besitzen muss, stehen hier die abhängigen Services, weil genau dieses Interface zum Aufruf eines Services dient und daher als Bibliothek eingebunden werden muss. Die Namensgebung dieser Bibliothek enthält auch die gewünschten Informationen über die Version der eingebundenen Services.

Somit muss nur diese Datei mit dem Parser verarbeitet werden, um anhand des jeweiligen Namens, sowie der Version, feststellen zu können, welche anderen Services der gerade betrachtete aufruft. Der Nachteil des Verfahrens ist jedoch, dass immer nur der letzte Build auf diese Weise geprüft werden kann, da der Build-Server immer nur die temporären Dateien des letzten Builds für jeden Service speichert. Somit können für ältere Builds keine Abhängigkeiten mehr erstellt werden, da die erforderlichen Dateien zu diesem Zeitpunkt nicht mehr vorliegen. Deswegen ist es angeraten, möglichst bei jedem Build die Abhängigkeiten neu zu analysieren. Abbildung 6-3 zeigt einen Auszug einer solchen *.classpath* Datei.

```
<classpathentry kind="src" path="src/conf/impl"/>
<classpathentry kind="src" path="src/conf/generated"/>
<classpathentry kind="src" path="src/java/impl"/>
<classpathentry kind="src" path="src/java/generated"/>
<classpathentry excluding="**" kind="src" path="src/model"/>
<classpathentry excluding="**" kind="src" path="src/process"/>
<classpathentry kind="src" output="build/test_local" path="test/local"/>
<classpathentry kind="src" output="build/test_remote" path="test/remote"/>
<classpathentry kind="src" output="build/testsuite" path="test/suite"/>
<classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
<classpathentry kind="con" path="org.eclipse.ajdt.core.ASPECTJRT_CONTAINER"/>
<classpathentry kind="con" path="org.eclipse.jdt.junit.JUNIT_CONTAINER/3"/>
<classpathentry kind="con" path="org.eclipse.jdt.USER_LIBRARY/ATP-1.0.0"/>
<classpathentry kind="lib" path="/atp-global/atp-common-lib/log4j/1.2.15/log4j-1.2.15.jar"/>
<classpathentry kind="lib" path="/atp-global/atp-common-lib/jbpm/3.2.3-atp-patched/jbpm-jpdl-atp-patched.j
<classpathentry kind="lib" path="/atp-global/atp-common-lib/atp-commons/commons.util/1.6.0/commons.util-1.
<classpathentry kind="lib" path="/atp-global/atp-common-lib/atp-commons/commons.ejb.util/1.4.0/commons.ejb
<classpathentry kind="lib" path="/atp-global/atp-client-lib/addresscheck/addresscheck-client-0.7.1.jar"/>
<classpathentry kind="lib" path="/atp-global/atp-client-lib/telno/telno-client-1.3.0.jar"/>
<classpathentry kind="lib" path="/atp-global/atp-common-lib/jconfig/2.9/jconfig.jar">
        <attributes>
                <attribute name="javadoc_location" value="http://www.jconfig.org/javadoc"/>
        </attributes>
</classpathentry>
<classpathentry kind="output" path="build/classes"/>
```

Version

Abhängige Client Bibliotheken

Abbildung 6-3: Auszug einer *.classpath* Datei

Diese nun festgestellten Abhängigkeiten werden in die vorher definierte Datenstruktur für jeden Service eingefügt und persistiert. Hierdurch baut sich mit der Zeit ein sehr genaues Abbild der Abhängigkeiten auf, welches sich zu ihrer Auflösung benutzen lässt.

6.3. Abhängigkeiten auflösen

Das automatische Erkennen von Abhängigkeiten der Services untereinander reicht jedoch noch nicht aus. Durch das Verfahren sind nur die direkten Abhängigkeiten in der Datenbank vorhanden. Es kann jedoch auch zu indirekten Abhängigkeiten kommen, die ebenfalls identifiziert werden müssen.

Im Folgenden wird ein Beispiel für eine indirekte Abhängigkeit gegeben. Ein Service A hat als Abhängigkeit den Service B und C. Service C hat wiederum einen weiteren Service D als Abhängigkeit. Da ein Service alle seine Abhängigkeiten benötigt, um einwandfrei zu funktionieren, würde es natürlich nicht reichen B und C zu installieren. C würde noch D benötigen und könnte somit nicht fehlerfrei laufen. Wenn C nicht funktioniert läuft A auch nicht. Dies bedeutet A hat D als indirekte Abhängigkeit. In Abbildung 6-4 wird dieser Zusammenhang noch einmal veranschaulicht.

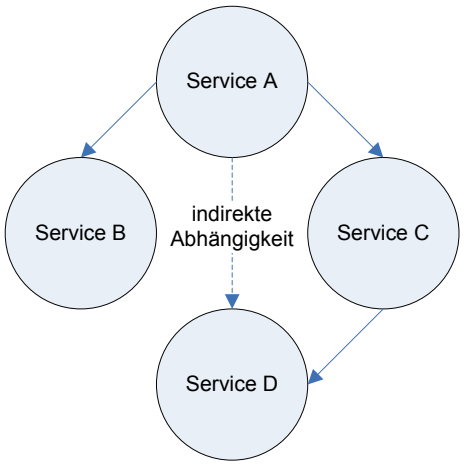


Abbildung 6-4: Beispiel von indirekten Abhängigkeiten von Services

Aus dieser Tatsache ergibt sich also ein ganzer Abhängigkeitsbaum, den es korrekt zu erstellen gilt, um auch alle indirekten Abhängigkeiten aufzulösen. Außerdem ist zu beachten, dass jede Version eines Services seine eigene Abhängigkeitsliste haben kann.

Wenn nun ein Abhängigkeitsbaum aufgebaut werden soll, gibt es zwei Probleme zu beachten. Erstens könnte es zu Zirkelbezügen in den Abhängigkeiten kommen. Betrachtet man zum Beispiel Abbildung 6-5, so sieht man, dass A von B abhängig ist, B von C und C wiederum von A.

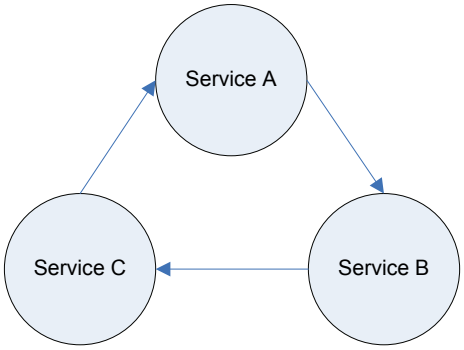


Abbildung 6-5: Beispiel einer Zirkel Abhängigkeit

Zweitens könnte ein Service doppelt in einem Baum vorkommen wenn es Abhängigkeiten von verschiedenen Versionen gibt. Ein Beispiel hierfür ist in Abbildung 6-6 aufgeführt.

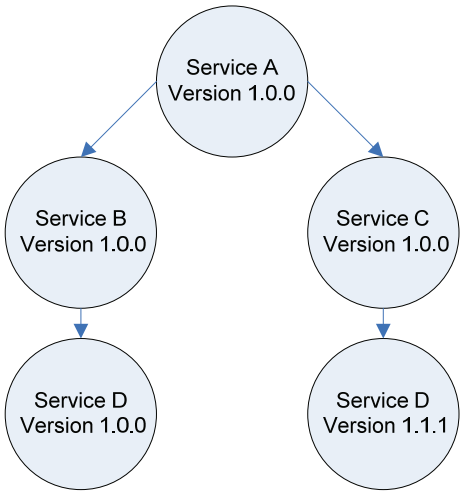


Abbildung 6-6: Beispiel für doppeltes Auftreten eines Services

Im abgebildeten Beispiel würde Service D zweimal im Baum auftauchen und zwar einmal in der Version 1.0.0 und einmal in der Version 1.1.1. Dies ist natürlich nicht gewünscht, da immer nur eine Version eines Services installiert sein kann. In der Praxis stellt dies jedoch kein großes Problem dar, da jede höhere Version abwärtskompatibel ist. Dadurch wird das Problem reduziert, da vereinfachend die höhere Version behalten und die niedrigere Version verworfen wird.

Dies lässt sich durch einen einfachen Algorithmus lösen. Beginnend mit dem Startknoten werden die direkten Abhängigkeiten betrachtet und an den Baum angehängt. Nun geht der Algorithmus eine Ebene tiefer und schaut sich wiederum die weiteren Abhängigkeiten an. Bei jedem Knoten der angehängt werden soll, wird geprüft, ob ein Knoten mit diesem Service bereits im vorliegenden Baum vorhanden ist. Sollte dies der Fall sein, werden die Versionsnummern verglichen. Ist der einzufügende Knoten kleiner als der bereits vorhandene, so wird er nicht zugefügt und der Algorithmus fährt fort. Ist der einzufügende

Knoten von der Version her größer, so muss der bereits im Baum vorhandene Knoten gelöscht werden und der mit der höheren Versionsnummer verknüpfte Knoten wird an den Baum angehängt.

Hierbei ist jedoch zu beachten, dass das Löschen eines Knotens auch die Löschung des kompletten Teilbaums dieses Knotens bedingt, da sich der Teilbaum durch das Einfügen einer anderen Version wieder grundlegend verändern kann.

Durch den vorgestellten Algorithmus wird auch das Problem von Zirkelabhängigkeiten umgangen, da jedes Mal der ganze Baum nach bereits vorhandenen Services untersucht wird und dieser in dem Fall durch eine höhere Version ersetzt wird, aber nie durch eine niedrigere. Hiermit ist sichergestellt, dass der Algorithmus terminiert und nicht in eine Endlosschleife läuft.

Zur besseren Darstellung wird im Folgenden ein Beispiel Schritt für Schritt durchlaufen.

In Abbildung 6-7 werden alle für das Beispiel relevanten Abhängigkeiten dargestellt. Ziel ist es, einen kompletten Abhängigkeitsbaum für Service A in Version 1.0.0 zu erstellen.

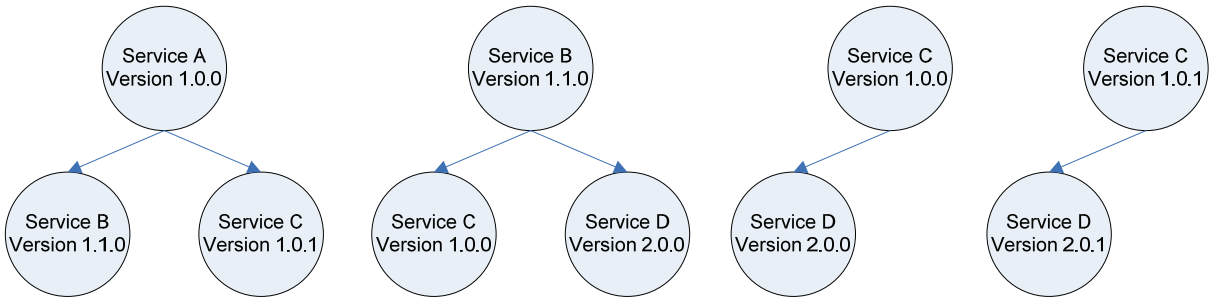


Abbildung 6-7: Abhängigkeiten Übersicht

Im ersten Schritt (Abbildung 6-8) wird Service A in Version 1.0.0 ausgewählt und dem noch leeren Baum zugefügt. Er stellt somit die Wurzel des Baumes dar.

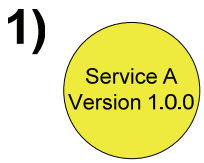


Abbildung 6-8: Algorithmus Beispiel Schritt 1

Als nächstes (Abbildung 6-9) wird nun versucht die Abhängigkeiten von Service A dem Baum zuzufügen. Da beide Abhängigkeiten noch nicht im Baum vorhanden sind, werden sie einfach an den Wurzelknoten von Service A angehängt.

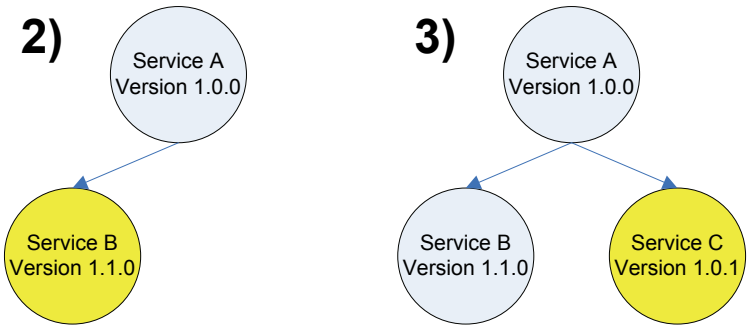


Abbildung 6-9: Algorithmus Beispiel Schritt 2 und 3

Nun werden die Abhängigkeiten von Service B betrachtet. Theoretisch hätte Service B als Abhängigkeit Service C in Version 1.0.0. Da jedoch Service C bereits in einer höheren Version vorhanden ist, wird diese Abhängigkeit nicht dem Baum zugefügt. (Abbildung 6-10)

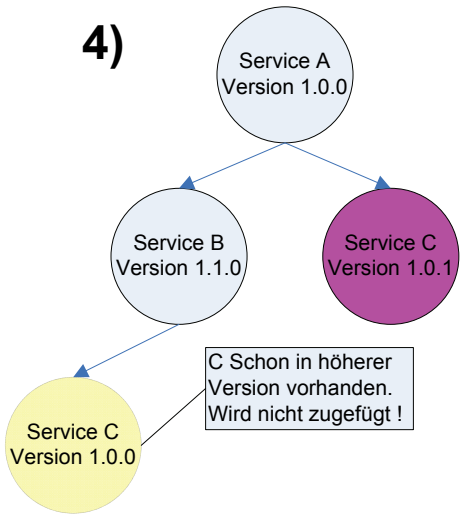


Abbildung 6-10: Algorithmus Beispiel Schritt 4

Der zweite abhängige Knoten D wird einfach dem Baum hinzugefügt, da er noch nicht vorhanden ist. (Abbildung 6-11)

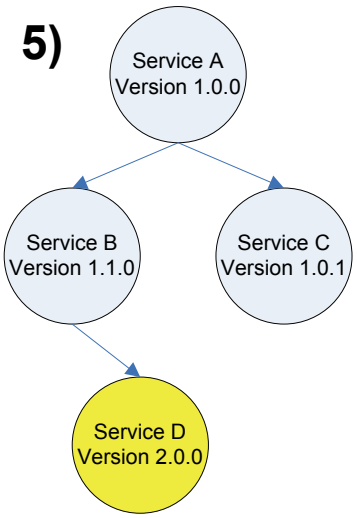


Abbildung 6-11: Algorithmus Beispiel Schritt 5

Anschließend werden die Abhängigkeiten von Knoten C betrachtet. Dieser benötigt Service D in Version 2.0.1, im Baum ist bis jetzt aber nur Version 2.0.0 vorhanden. Daher muss der bereits vorhandene Knoten und sein Teilbaum aus dem Baum gelöscht und die höhere Version angehängt werden. (Abbildung 6-12)

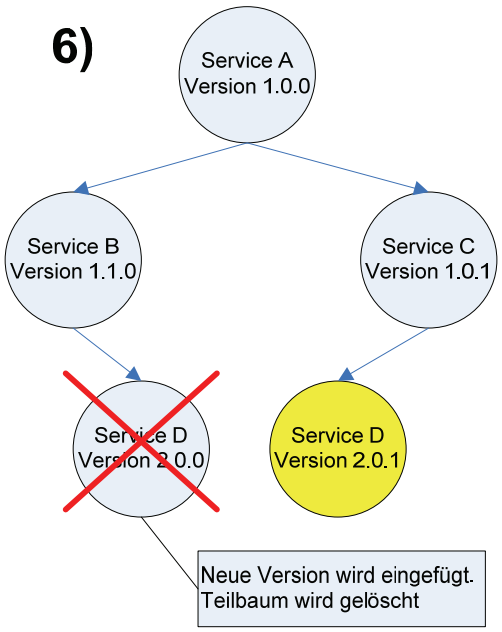


Abbildung 6-12: Algorithmus Beispiel Schritt 6

Da nun keine weiteren Abhängigkeiten vorhanden sind, ergibt sich für dieses Beispiel der Abhängigkeitsbaum für Service A in Version 1.0.0 folgendermaßen (Abbildung 6-13):

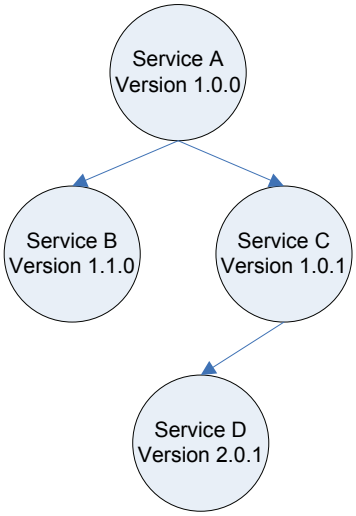


Abbildung 6-13: Endzustand des Algorithmus Beispiels

Der vorgestellte Algorithmus wurde in Java implementiert und dient dem fertigen Prototypen zur Auflösung von Abhängigkeiten.

6.4. Laden der Services vom Build-Server

Nachdem nun ein kompletter Abhängigkeitsbaum entstanden ist, müssen die entsprechenden Versionen zur weiteren Bearbeitung vom Build-Server geladen werden. Hierzu wird in der Datenbank einfach nach den entsprechenden Services in der benötigten Version gesucht. Da in der Datenbank auch die Adresse zum Herunterladen geparst wurde, bereitet es keine Schwierigkeit, das entsprechende Jar-Archiv herunterzuladen und zur weiteren Verarbeitung der Konfiguration zur Verfügung zu stellen.

6.5. Konfiguration der Services

6.5.1. Konfigurationsdateien bearbeiten

Die automatische Konfiguration der heruntergeladenen Services ist ein wichtiges Merkmal, da es ein sehr zentraler Bestandteil der Installation ist. Sollte ein Service vor seiner Installation nicht richtig konfiguriert werden, so ist nicht absehbar, ob er wunschgemäß funktionieren wird. Die Konfiguration kann hierbei sehr unterschiedlich ausfallen: bei einem Service kann es nötig sein, den Deployment Descriptor anzupassen, bei einem anderen müssen eventuell verschiedenste Einträge in einer anderen XML-Datei angepasst werden. Eine Kombination von mehreren dieser Möglichkeiten ist hierbei auch möglich.

Da bei den hier betrachteten Services lange das Konfigurations-Framework jConfig[17] eingesetzt wurde, ist meistens die Datei *config.xml* anzupassen, die zur Konfiguration durch das Framework benutzt wird. In Abbildung 6-14 ist ein Beispiel dieser Konfigurationsdatei zu sehen.

```
- <properties>
  - <category name="BankAccountCheck">
      <property name="bankAccountCheckServerUrl" value="http://www.jconfig.org/"/>
    </category>
  </properties>
```

Abbildung 6-14: Beispiel einer *config.xml* Datei des jConfig Frameworks

Solche XML Dateien können jedoch beliebig komplex verschachtelt sein. Auch die Werte, die verändert werden müssen, können sich unterscheiden. Wenn dies manuell bei einer Installation gemacht wird, ist das erste Problem, dass die zu editierenden Dateien innerhalb des Jar-Archivs abgelegt sind. Dies bedeutet, dass zunächst das Archiv entpackt werden muss, um danach die Konfigurationsdateien zu editieren. Anschließend wird das Ganze wieder in ein Jar-Archiv verpackt, welches man dann dem Applikationsserver zur Installation übergeben kann.

Zur Automatisierung dieses Prozesses wurde ein eigenes Konzept entwickelt. Es wurde zunächst ein Interface `FileManipulater` definiert, welcher eine zu implementierende Methode `manipulate` vorschreibt. Sie erhält einen `InputStream`, also die zu manipulierende Datei als Eingabestrom, und gibt wiederum einen veränderten Eingabestrom zurück (siehe Abbildung 6-15).

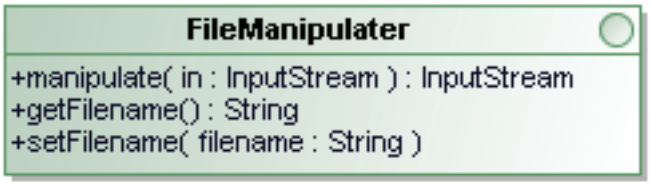


Abbildung 6-15: FileManipulater Interface

Der `FileManipulater` dient als Filter, der zwischen den Eingabestrom geschaltet wird. Durch dieses Konzept ändert sich für nachfolgende Prozesse nichts an den erwarteten Eingabeparametern. Erwartet also ein nachfolgender Prozess einen Eingabestrom, so

[17] http://www.jconfig.org/

bekommt er diesen auch weiterhin geliefert. Abbildung 6-16 stellt diesen Sachverhalt noch einmal bildlich dar.

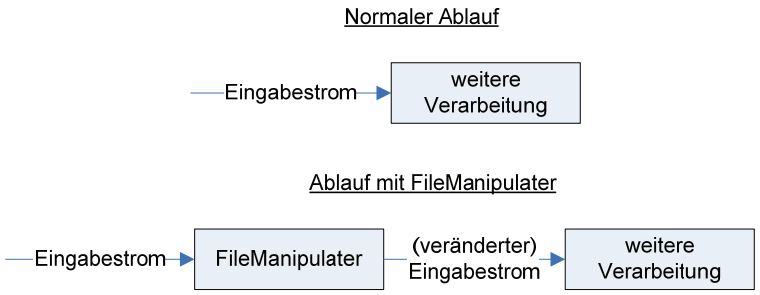


Abbildung 6-16: Funktionsweise eines FileManipulaters

Außerdem wurden zwei konkrete Klassen geschrieben, die dieses Interface implementieren, `JarFileManipulater` sowie `XmlFileManipulater`.

Der `JarFileManipulater` ist im Grunde ein Container für andere Klassen, die das `FileManipulater` Interface implementieren. Er übernimmt die Aufgabe, ein Archiv zu entpacken und für alle, durch weitere Manipulatoren bestimmte Dateien, wiederum den entsprechenden Manipulator aufzurufen. Es können also beliebig viele Dateien innerhalb eines Jar-Archives angegeben und verändert werden. Abschließend verpackt er alles wieder in ein Archiv, welches er als Eingabestrom zurückgibt. Durch dieses Konzept ist es möglich, selbst tief verschachtelte Konstruktionen wie ein Jar-Archiv in einem Jar-Archiv zu verändern.

Die zweite Klasse `XmlFileManipulater` dient zur Manipulation von XML-Dateien. Es ist möglich, ihr beliebig viele XML-Operationen zuzuordnen, die auf eine XML-Datei angewendet werden. Die XML-Operationen sind hierbei recht einfach definiert: Sie enthalten einen XPath[18] Ausdruck, um den oder die Knoten in dem XML Dokument zu selektieren, welche verändert werden sollen, sowie den Wert, der für den selektierten Knoten zu setzen ist.

Es gibt drei verschiedene Änderungsmodi:

1. Mit REPLACE wird der Wert von dem oder den gewählten Knoten einfach ersetzt. Hier ist jedoch die Vorraussetzung, dass der oder die Knoten bereits im XML Dokument existieren. In Abbildung 6-17 wird ein Beispiel für diese Operation gezeigt.

[18] XPath ist eine Abfragesprache, die dazu dient Teile eines XML-Dokuments zu addressieren.

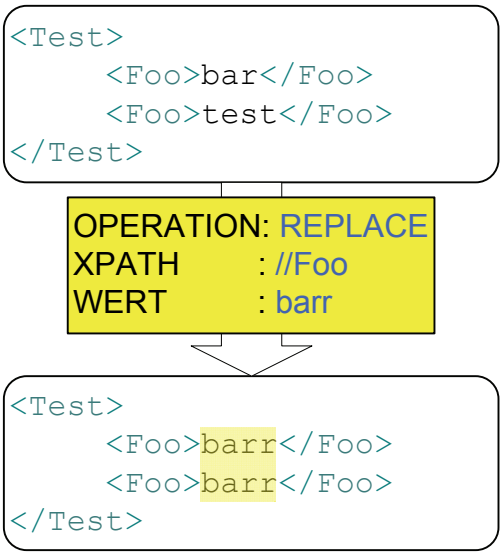


Abbildung 6-17: Beispiel für die Operation REPLACE

2. ADD führt im Grunde eine identische Operation aus, jedoch werden die fehlenden Knoten des XPath Ausdrucks bei Bedarf erzeugt. Wichtig ist hierbei, dass nur ein einzelner Knoten über XPath selektiert wird, da an einem XPath Ausdruck nicht erkannt werden kann, an welcher Stelle mehrere Knoten eingefügt werden müssten. In Abbildung 6-18 wird die Ausführung einer solchen Operation dargestellt.

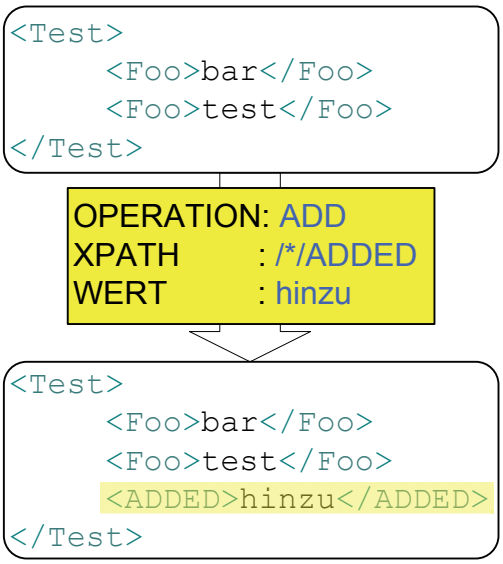


Abbildung 6-18: Beispiel für die Operation ADD

3. Schließlich löscht der Modus DELETE den oder die gewählten Knoten aus dem XML-Baum (Abbildung 6-19).

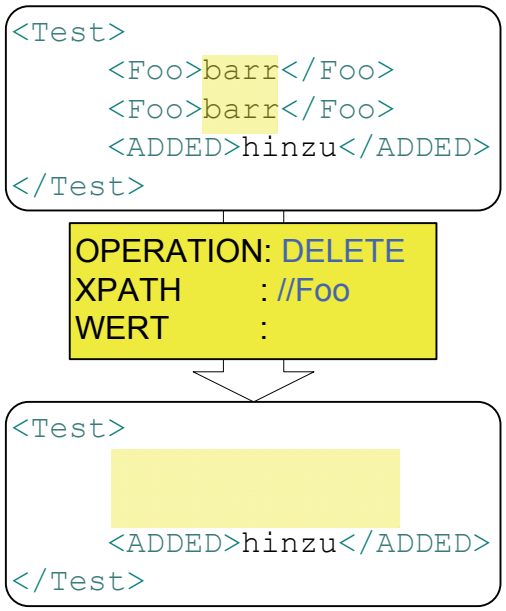


Abbildung 6-19: Beispiel für die Operation DELETE

In Tabelle 6-1 werden alle verfügbaren Modi noch einmal zusammengefasst.

Modus	Mehrere Knoten selektierbar	Beschreibung
ADD	Nein	Ändert den Wert eines einzelnen Knoten und legt eventuell nicht vorhandene Knoten des *XPath* Pfades an.
REPLACE	Ja	Ändert den Wert eines oder mehrerer Knoten die durch den *XPath* Ausdruck gewählt wurden.
DELETE	Ja	Löscht den oder die durch *XPath* selektierten Knoten aus dem XML-Dokument.

Tabelle 6-1: Modi der XMLOperation

Durch die Verwendung von XPath ist es auf einfache Weise möglich selbst komplex aufgebaute XML-Dateien zu verändern, da mittels XPath jeder beliebige Knoten in einem XML-Dokument selektiert werden kann.

Mit diesen beiden Klassen ist es möglich, das heruntergeladene Jar-Archiv zu manipulieren, beziehungsweise die nötigen Konfigurationen zu beschreiben. Eine komplette Übersicht über die implementierten Klassen gibt das Klassendiagramm in Abbildung 6-20

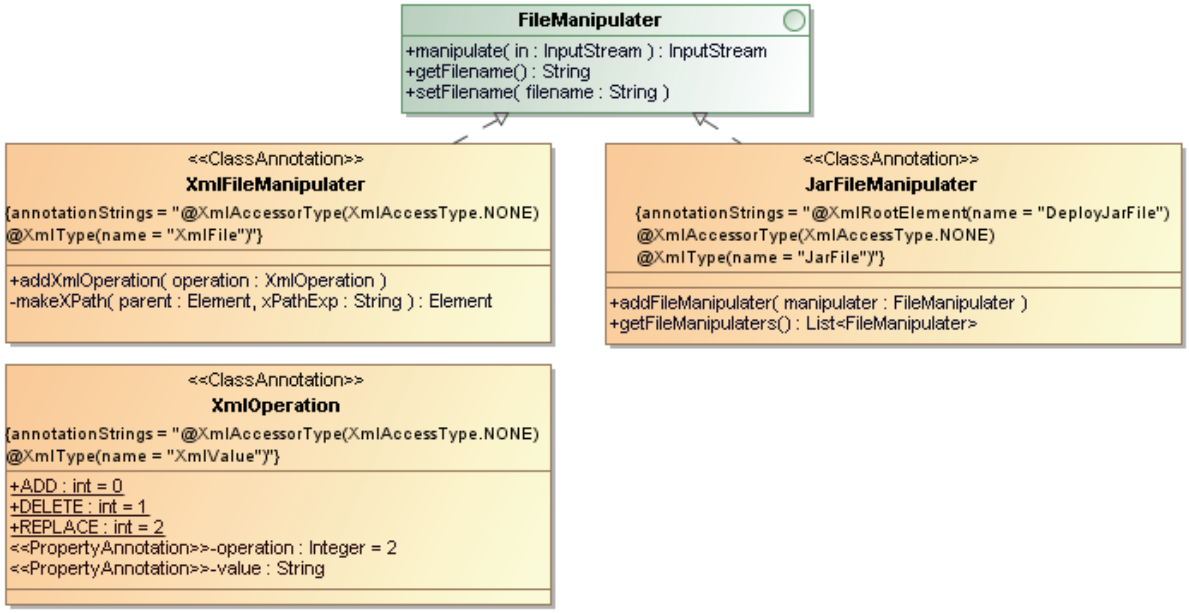


Abbildung 6-20: Klassendiagramm der FileManipulater und XMLOperationen

Durch dieses einfache und sehr offene Konzept ist es möglich, die Struktur für weitere Dateitypen zu erweitern. Sollte es nötig sein, andere Dateien zu editieren, ist nur ein Manipulator für die entsprechende Datei zu schreiben, welcher das Interface `FileManipulater` implementiert.

Nachdem nun die implementierten Klassen vorgestellt wurden, muss dieses Konzept natürlich dem Benutzer mit einer geeigneten Schnittstelle zur Verfügung gestellt werden.

Um es dem Benutzer so einfach und verständlich wie möglich zu machen, wurde eine einfache XML-Syntax verwendet. In Abbildung 6-21 ist ein Beispiel für diese Syntax zu sehen.

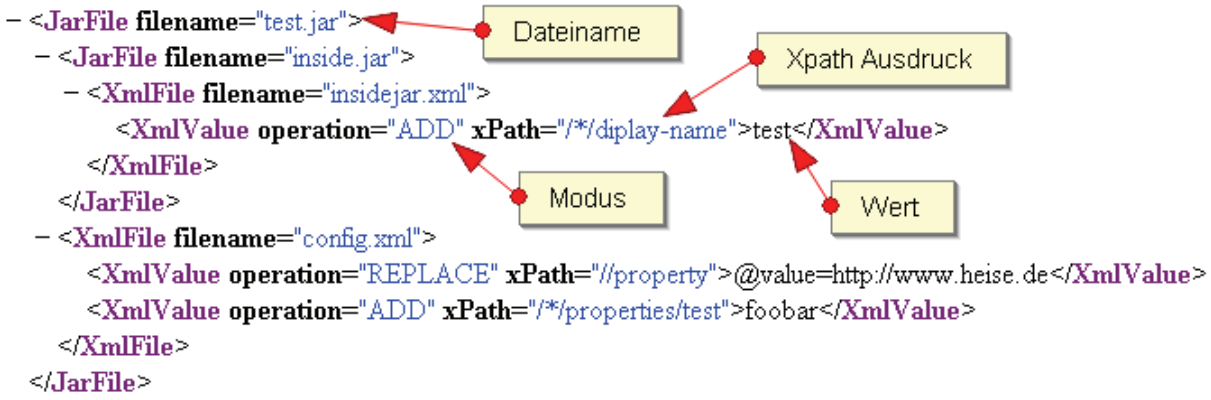


Abbildung 6-21: XML Syntax der XmlFileManipulater und JarFileManipulater

Ermöglicht wird diese Abbildung von Java Klassen auf XML Elemente durch *die Java Architecture for XML Binding* oder kurz JAXB. Durch diese ist es möglich, Java Objekte

durch einfache Annotationen nach XML zu serialisieren und auch wieder zu deserialisieren.

6.5.2. Anlegen von benötigten Ressourcen

Viele Services benötigen zur Inbetriebnahme verschiedene Ressourcen, welche im Applikationsserver angelegt werden müssen. Aber auch das Anlegen dieser Ressourcen soll automatisiert, beziehungsweise vereinfacht werden. Wie in Abbildung 6-22 zu sehen, wurde zunächst ein Interface für den sogenannten `RessourceCreator` erstellt.

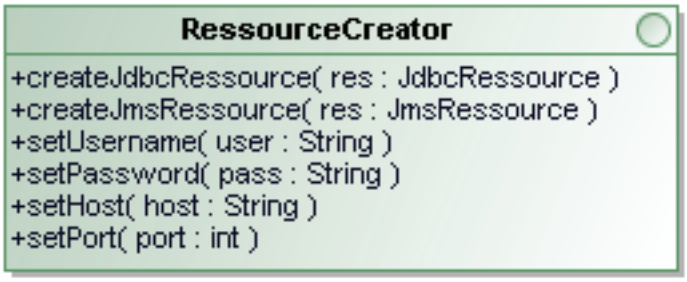


Abbildung 6-22: RessourceCreator Interface

Durch die Interface Definition bleibt die konkrete Implementierung offen für verschiedene Ansätze um Ressourcen anzulegen. Der `RessourceCreator` definiert die Methoden zum Anlegen von JDBC- und JMS-Ressourcen, sowie das Definieren des zu verwendenden Servers mit seinem Benutzernamen und Passwort. Zur Instanziierung eines `RessourceCreators` wurde eine Fabrik[19] geschrieben, welche über eine Property-Datei die zu instanziierende Klasse festlegt.

Hierdurch ist es ohne Probleme möglich, die implementierende Klasse zur Laufzeit zu ändern, ohne den Quellcode zu verändern. Für den Prototyp wurde eine auf JMX basierende implementierende Klasse geschrieben. Über JMX steht eine standardisierte Methode zur Verfügung, einen Applikationsserver zu verwalten, um zum Beispiel Ressourcen anzulegen. Die Ressourcen selber wurden wiederum in eigenen Klassen implementiert. In Abbildung 6-23 ist das Klassendiagramm der Ressourcen zu sehen.

[19] Eine Fabrik ist ein Software Entwurfsmuster, welches besagt das ein Objekt nicht direkt erstellt wird, sondern durch eine zweite Klasse, eine so genannte Fabrik. Hierdurch können im Quellcode Interfaces benutzen werden, da die eigentliche Implementierung bis zur Laufzeit offen bleibt.[DPattern04]

Abbildung 6-23: Klassendiagramm der Ressourcen Klassen

Auch hier wurde JAXB zur Serialisierung und Deserialisierung in ein einfach lesbares XML-Format verwendet. Eine XML-Datei für Ressourcen ist in Abbildung 6-24 zu sehen.

```
- <Ressources>
  - <JdbcRessource JndiName="jdbc/test">
      <PoolName>test</PoolName>
      <RessourceType>javax.sql.XADataSource</RessourceType>
    - <Properties>
        <Property value="test" name="User"/>
        <Property value="testest" name="Password"/>
        <Property value="jdbc:oracle:thin:@test.de:1752:TESTT1" name="URL"/>
      </Properties>
      <DataSourceClassName>oracle.jdbc.xa.client.OracleXADataSource</DataSourceClassName>
    </JdbcRessource>
  - <JmsRessource type="FACTORY" JndiName="jms/testFactory">
      <PoolName>testFactory</PoolName>
      <RessourceType>javax.jms.QueueConnectionFactory</RessourceType>
      <Properties/>
    </JmsRessource>
  - <JmsRessource type="DESTINATION" JndiName="jms/testQueue">
      <PoolName>testQueue</PoolName>
      <RessourceType>javax.jms.Queue</RessourceType>
      <Properties/>
    </JmsRessource>
  </Ressources>
```

Abbildung 6-24: XML Syntax der Ressourcen

6.5.3. Konfiguration sichern

Mit den beiden oben vorgestellten Konzepten der Datei-Manipulation und der Anlage von Ressourcen lässt sich nun eine komplette Konfiguration beschreiben, die zum Installieren eines Services nötig ist. Es können Dateien bei der Installation verändert oder Ressourcen angelegt werden. Jedoch ergebe es keinen Mehrwert, wenn dies jedes Mal vor einer Installation neu beschrieben werden müsste. Aus diesem Grund sollte eine solche Konfiguration gesichert werden können, da sich meist nur geringe Teile ändern. So werden beispielsweise die Ressourcen immer von einem Service benötigt. Hier ändert sich, je nachdem auf welchem Server installiert wird, höchstens die Zuordnung des Datenbankservers. Ähnlich sieht es mit Dateien aus, da sich hier meist nur umgebungsbezogene Einstellungen ändern. Um diesem Umstand Rechnung zu tragen, wurde ein vierstufiges Konfigurationssystem implementiert. Die Konfigurationsdateien sind hierbei immer gleich aufgebaut: es sind XML-Dokumente, welche aus den beiden bereits vorgestellten XML-Strukturen, der FileManipulater und Ressourcen zusammengesetzt wurden. Abbildung 6-25 zeigt eine fertig zusammengesetzte Konfigurationsdatei.


```
- <Configuration>
  - <Ressources>
    - <JdbcRessource JndiName="jdbc/test">
        <PoolName>test</PoolName>
        <RessourceType>javax.sql.XADataSource</RessourceType>
      - <Properties>
          <Property value="test" name="User"/>
          <Property value="testest" name="Password"/>
          <Property value="jdbc:oracle:thin:@test.de:1752:TESTT1" name="URL"/>
        </Properties>
        <DataSourceClassName>oracle.jdbc.xa.client.OracleXADataSource</DataSourceClassName>
      </JdbcRessource>
    - <JmsRessource type="FACTORY" JndiName="jms/testFactory">
        <PoolName>testFactory</PoolName>
        <RessourceType>javax.jms.QueueConnectionFactory</RessourceType>
        <Properties/>
      </JmsRessource>
    - <JmsRessource type="DESTINATION" JndiName="jms/testQueue">
        <PoolName>testQueue</PoolName>
        <RessourceType>javax.jms.Queue</RessourceType>
        <Properties/>
      </JmsRessource>
  </Ressources>
  - <DeployJarFile>
    - <XmlFile filename="config.xml">
        <XmlValue operation="REPLACE" xPath="//property">@value=http://www.heise.de</XmlValue>
        <XmlValue operation="ADD" xPath="/*/properties/test">foobar</XmlValue>
      </XmlFile>
  </DeployJarFile>
</Configuration>
```

Abbildung 6-25: XML Syntax einer fertigen Konfigurationsdatei

Für jeden Service können mehrere Konfigurationen gespeichert werden, die unterschiedliche Prioritäten besitzen. Zunächst kann jeder Service eine allgemeine Konfiguration haben, die die erste Konfigurationsstufe darstellt. In der zweiten Stufe kann jeder Service eine Konfiguration für eine bestimmte Version haben. Die dritte Stufe definiert die Konfiguration für einen Service auf einem bestimmten Server. Schließlich ist die vierte und letzte Stufe für die Konfiguration einer Version auf einem bestimmten Server gedacht. In Tabelle 6-2 sind die vier Konfigurationsstufen noch einmal zusammengefasst.

Stufe	Service Konfiguration	Version Konfiguration	Server Konfiguration	Beschreibung
4	-	**X**	**X**	Konfiguration einer bestimmten Version eines Services für einen bestimmten Server
3	**X**	-	**X**	Service Konfiguration für einen bestimmten Server
2	-	**X**	-	Allgemeine Konfiguration einer bestimmten Version eines Services
1	**X**	-	-	Allgemeine Service Konfiguration

Tabelle 6-2: Übersicht der verschiedenen Konfigurationsstufen

Bei diesem Konzept überschreibt jede höhere Stufe alle niedrigeren. Damit ist es möglich, für jeden Service eine Grundkonfiguration anzulegen. Dies würde der Stufe eins entsprechen. Diese Konfiguration könnte jedoch durch eine für einen bestimmten Server, auf dem installiert wird, überschrieben werden, was Stufe drei entsprechen würde.

Durch diese vier Stufen ist eine flexible Konfiguration für jeden Service möglich, die in der Regel nur einmal durchgeführt werden muss und danach reproduzierbar funktioniert. Hierdurch wird der Konfigurationsaufwand im Idealfall auf einen einmaligen Durchlauf reduziert.

6.6. Installation / Deployment der Services

Nachdem alle nötigen Vorbereitungen für die eigentliche Installation getroffen wurden, indem alle Dateien angepasst und Ressourcen angelegt wurden, muss noch der eigentliche Installationsvorgang angestoßen werden. Hierzu wurde eine Java-Klasse geschrieben, welche auf dem JSR88 aufsetzt. Der JSR88 beschreibt eine einheitliche Schnittstelle für Applikationsserver, um ein Deployment durchzuführen. Durch diese Schnittstelle ist eine einfache Möglichkeit zum Installieren von Services gegeben. Auch hier wurde auf Flexibilität großen Wert gelegt. Das heißt, die Klasse zum Deployen `Jsr88Deployer` wurde gegen ein eigenes Interface implementiert. Dies ermöglicht den einfachen Wechsel dieser Klasse gegen eine andere, sollte beispielsweise die JSR88 Spezifikation nicht von dem verwendeten Applikationsserver unterstützt werden. Da die `Deployer` Klasse von

einer Fabrik erzeugt wird, ist das einfache Ersetzen zur Laufzeit ebenso möglich. In Abbildung 6-26 wird das Interface sowie die implementierende Klasse als Klassendiagramm dargestellt.

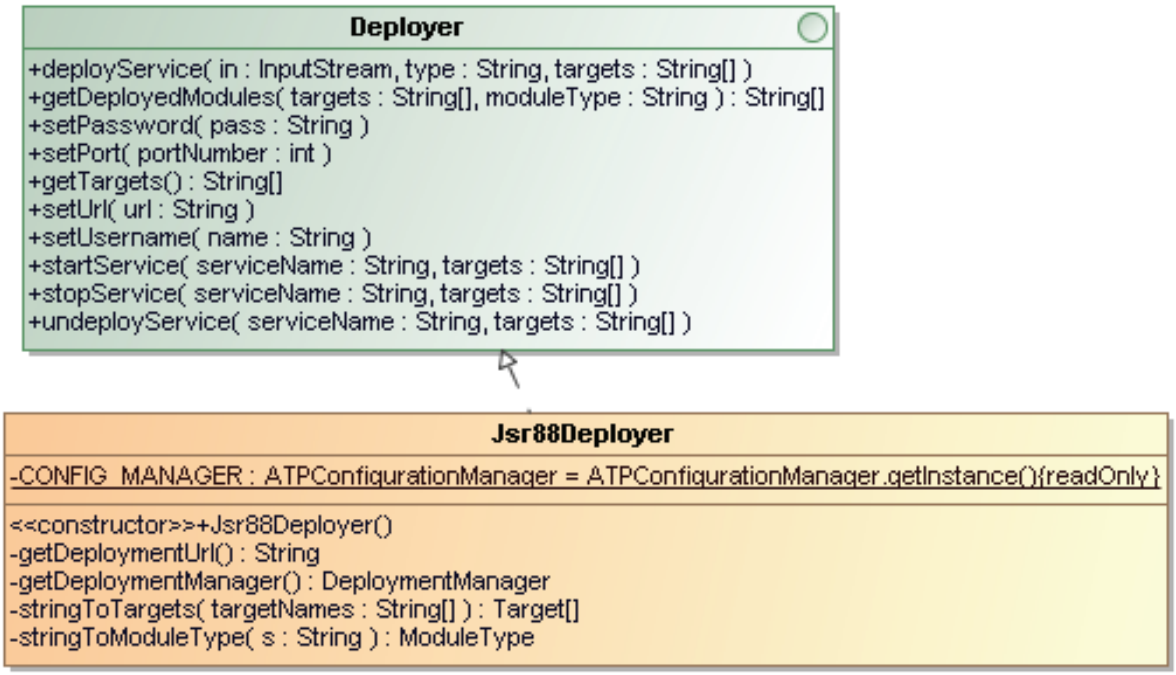


Abbildung 6-26: Klassendiagramm des Deployer Interfaces und des Jsr88Deployers

6.7. Erstellen der History

Eine der letzten Anforderungen der automatischen Installationsroutinen war es, eine Historisierung über die Installationsprozesse zu führen. Hier wurde im manuellen Verfahren eine Wiki Seite gepflegt, welche diese Informationen vorgehalten hat. Dieses Konzept wurde durch den im Laufe dieser Arbeit entwickelten Service ersetzt. Hierzu wurde die Datenstruktur erweitert, welche in der Datenbank gespeichert wurde. In der Datenbank wird nun für jeden Server bei jedem Installations- oder Deinstallationsvorgang vorgehalten, welche Services nach der erfolgreichen Operation auf dem Server vorhanden sind, beziehungsweise waren. Hierdurch ist es möglich, für jeden Server den Zustand für jeden beliebigen Zeitpunkt zu bestimmen. Diese Möglichkeit geht weit über die Möglichkeit des manuellen Wiki-Eintrages hinaus. Außerdem wird so bei einem Fehler der Services ermöglicht, eine alte funktionierende Konstellation nachzubilden.

Damit dieses Prinzip jedoch funktioniert, müssen die Server, auf die installiert werden soll, in die Datenbank aufgenommen werden. Aus diesem Grund wurde die Datenstruktur um die Speicherung des Servers erweitert. Dies bedeutet für den Service, dass er nur auf

vorher definierte Server installieren kann. Durch dieses Verfahren wird gewährleistet, dass der Server unter einem Namen bekannt ist und so eine eindeutige Zuordnung für die Historisierung besteht. Abbildung 6-27 zeigt die komplette Persistenzschicht mit der um die Speicherung der Server, Konfigurationen und Historisierung erweiterten Klassenstruktur.

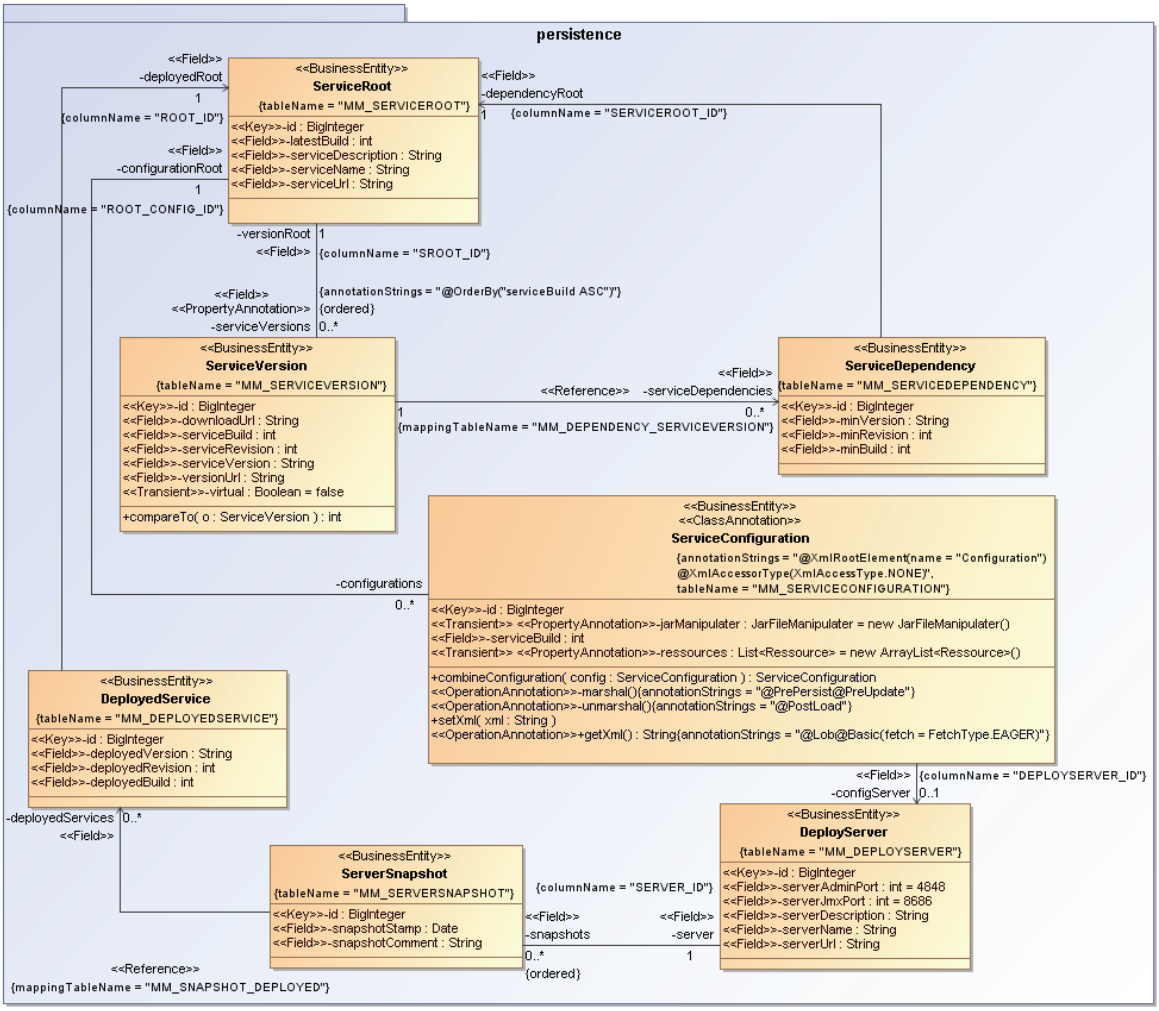


Abbildung 6-27: Persistenzschicht des Installationsservice

6.8. *Der Service Client*

Da es sich bei der Umsetzung zunächst selber um einen Service handelt, wird für die Benutzung ein Client benötigt. Der Service selbst stellt nur Funktionen zur Verfügung, die von einem Client angestoßen werden müssen.

Der Client wurde als Kommandozeilenclient in Java implementiert. Er kann auf alle vom Service bereitgestellten Funktionen zugreifen und ermöglicht so das automatische Installieren von Services von der Kommandozeile aus.

Der Client unterstützt durch mehrere einstellbare Parameter eine Vielzahl von Installationsszenarien. So ist es möglich, die bereits angesprochenen Konfigurationsdateien der Services zuzuordnen oder aber sich anzeigen zu lassen. Des Weiteren ist es möglich, jeden beliebigen Service, der auf dem Build-Server vorliegt, zu installieren. Auch die History so wie alle zum Deployment möglichen Server sind auf einfache Weise abrufbar. Abbildung 6-28 zeigt einen Aufruf der Hilfe des Clients und listet alle verfügbaren Befehle auf.

```
D:\dev\installer_cli>atpget -h

usage: [options] command [command parameter]
 -h,--help <[command]>   show help for command

Commands:
  addserver              add or change deploy server
  dependencies           list service dependencies
  deployed               list deployed services
  directdependencies     list only direct service dependencies
  getconfig              get the config for services
  install                install services
  listserver             list available deploy server
  setconfig              sets the config for services
  start                  start services
  stop                   stop services
  uninstall              uninstall services
  update                 update repository
```

Abbildung 6-28: Allgemeine Hilfe des Installationsservice Clients

Die wichtigsten zur Verfügung stehenden Befehle werden im Folgenden kurz erläutert.

6.8.1. Server hinzufügen / ändern

Durch den Befehl *addserver* lassen sich neue Server, auf die installiert werden kann, anlegen oder ändern. Wie in der Hilfe zu dem Befehl zu sehen ist, lassen sich alle Eigenschaften eines Servers über die Kommandozeile setzen. (Abbildung 6-29)

```
D:\dev\installer_cli>atpget -h addserver

usage: [options] addserver <servername> <serverurl> [<description>]
                 [<adminport>] [<jmxport>]
 -h,--help <[command]>   show help for command
```

Abbildung 6-29: Hilfe des Befehls *addserver* des Installationsservice Clients

6.8.2. Auflisten von Abhängigkeiten

Für das Auflisten der Abhängigkeiten stehen zwei Befehle zur Verfügung. Mit dem Befehl *dependencies* wird ein kompletter Abhängigkeitsbaum aufgebaut und angezeigt. Im Gegensatz dazu werden mit *directdependencies* lediglich direkte Abhängigkeiten eines Services angezeigt.

Ein Beispiel für die Ausgabe der Abhängigkeiten ist in Abbildung 6-30 zu sehen.

```
D:\dev\installer_cli>atpget dependencies centraflex

Servicename                          Version          REV   Build
-------------------------------------------------------------------------------

centraflex-candidate                   0.7         14447      47
customer-candidate                   0.3.1         13842       5
pbxif-candidate                        2.8         14079      20
contract-candidate                  0.15.5         13974      12
process-candidate                    1.1.0             0       0
product-candidate                    0.5.0         14403      20
order-candidate                      0.5.0         14182      28
atp_auth-candidate                   1.1.0             0       0
telno-candidate                      1.3.0          8282       2
addresscheck-candidate               0.7.1         13836       3
partner-candidate                    0.1.0             0       0
```

Abbildung 6-30: Ausgabe einer Abhängigkeitenliste des Installationsservice Clients

6.8.3. Zugriff auf die Historisierung

Auf die Historisierung des Installationsservice lässt sich über den Befehl *deployed* zugreifen. Wird *deployed* ohne Parameter aufgerufen, so wird der aktuelle Stand des angegebenen Servers angezeigt. Wird jedoch ein Datum und eine Uhrzeit definiert, so zeigt die Ausgabe den Stand des angegebenen Servers zu diesem Zeitpunkt. Abbildung 6-31 zeigt die Hilfe zu dem Befehl *deployed,* sowie einen Aufruf des Befehls für den Server *local* ohne Parameter.

```
D:\dev\installer_cli>atpget -h deployed

usage: [options] deployed [<dd.mm.yyyy/hh:mm>]
 -h,--help <[command]>   show help for command
 -s,--server <server>    Server to Use

D:\dev\installer_cli>atpget -s local deployed

Servicename                          Version          REV   Build
-------------------------------------------------------------------------------

billing-candidate                      0.3.3         13315      31
customer-candidate                     0.2.0          9202       3
order-candidate                        0.4.5         13304      20
atp_auth                               1.1.0         12329      18
addresscheck-candidate                 0.7.0          8872       2
pbxif-candidate                        2.7.0         13224      17
centraflex-candidate                   0.6.0         13314      41
telno-candidate                        1.3.0          8282       2
remedy                                 1.1.4         10773      11
contract-candidate                    0.15.2         13330       9
```

Abbildung 6-31: Ausgabe des Befehls *deployed* sowie dessen Hilfe des Installationsservice Clients

6.8.4. Konfiguration setzen / laden

Über den Befehl *getconfig* ist es möglich, die Konfiguration, welche für einen Service gesetzt wurde, lokal zu speichern. Hierbei bestimmen die Optionen, um welche Konfigurationsstufe es sich handelt. Je spezifischer die Optionen gewählt werden, desto eindeutiger wird auch die Konfiguration. Abbildung 6-32 zeigt die Hilfe des Befehls *getconfig* sowie einen Aufruf für die allgemeine Konfiguration des Services *Centraflex*.

```
D:\dev\installer_cli>atpget -h getconfig

usage: [options] getconfig <servicename>
 -b,--build <build>          service build to use
 -f,--file <filename>        config file to use
 -h,--help <[command]>       show help for command
 -r,--revision <revision>    service revision to use
 -s,--server <server>        Server to Use
 -t,--trunk                  use Trunk
 -v,--version <version>      service version to use

D:\dev\installer_cli>atpget -f config.xml getconfig centraflex

config successfully saved
```

Abbildung 6-32: Hilfe des Befehls *getconfig* des Installationsservice Clients

Der Befehl *setconfig* ist hierbei genau das Gegenteil und setzt die entsprechende Konfiguration für den gewählten Service(Abbildung 6-33).

```
D:\dev\installer_cli>atpget -h setconfig

usage: [options] setconfig <servicename>
 -b,--build <build>          service build to use
 -f,--file <filename>        config file to use
 -h,--help <[command]>       show help for command
 -r,--revision <revision>    service revision to use
 -s,--server <server>        Server to Use
 -t,--trunk                  use Trunk
 -v,--version <version>      service version to use
```

Abbildung 6-33: Hilfe des Befehls *setconfig* des Installationsservice Clients

6.8.5. Installation / Deinstallation von Services

Zum Installieren von Services wird der Befehl *install* verwendet. Wie aus der Hilfe in Abbildung 6-34 ersichtlich stehen hierfür eine Reihe Optionen zur Verfügung.

```
D:\dev\installer_cli>atpget -h install

usage: [options] install <servicename>
 -b,--build <build>                    service build to use
 -f,--file <filename>                  config file to use
 -h,--help <[command]>                 show help for command
 -l,--login <<username>:<password>>    glassfish login credentials
 -local                                Local Install Mode without Logging
 -r,--revision <revision>              service revision to use
 -s,--server <server>                  Server to Use
 -single                               Install Single Service
 -t,--trunk                            use Trunk
    --targets <target1,target2,...>    targets to deploy to
 -v,--version <version>                service version to use
```

Abbildung 6-34: Hilfe des Befehls *install* des Installationsservice Clients

So ist es möglich, über die Option `build`, `revision` und `version` exakt zu spezifizieren, welche Version eines Service genau installiert werden soll. Die `file` Option bietet die Möglichkeit, die vom Installationsservice gespeicherte Konfiguration durch eine eigene, für diese spezielle Installation erstellte, zu ersetzen. Mit der Option `single` wird nur der angegebene Service installiert. Ohne die Option `single` wird ein Abhängigkeitsbaum

erstellt, welcher genau anzeigt welche Services installiert werden, beziehungsweise welche schon auf dem Server vorhanden sind. Ein Beispiel einer solchen Ausgabe ist in Abbildung 6-35 zu sehen.

```
D:\dev\installer_cli>atpget -l admin:adminadmin -s local install centraflex

Server Details
##############################################################################################

Name        : local
Url         : localhost
Description : lokaerl

Installed Services
##############################################################################################

Servicename                    Version     REV   Build   installed
----------------------------------------------------------------------------------------------

centraflex-candidate             0.7     14447     47    (0.6.0_REV13586_B44)
contract-candidate             0.15.5    13974     12    (0.15.3_REV13574_B11)
order-candidate                 0.5.0    14182     28    (0.4.5_REV13304_B20)
pbxif-candidate                   2.8    14079     20    (2.7.0_REV13224_B17)
telno-candidate                 1.3.0     8282      2    (1.3.0_REV8282_B2)
addresscheck-candidate          0.7.1    13836      3    (0.7.0_REV8872_B2)

Unavailable dependent Services
##############################################################################################

Servicename                    Version     REV   Build   installed
----------------------------------------------------------------------------------------------

process-candidate               1.1.0        0       0
atp_auth-candidate              1.1.0        0       0
partner-candidate               0.1.0        0       0

Services to be installed
##############################################################################################

Servicename                    Version     REV   Build   installed
----------------------------------------------------------------------------------------------

customer-candidate              0.3.1    13842       5
product-candidate               0.5.0    14403      20

install listed Services (yes/no): _
```

Abbildung 6-35: Ausgabe des Befehls *install* des Installationsservice Clients

Zum Deinstallieren von Services steht der Befehl *uninstall* zur Verfügung. Er deinstalliert genau einen einzelnen Service von dem Applikationsserver.

6.8.6. Aktualisieren der Datenbank

Durch den Befehl *update* ohne weitere Parameter wird ein erneutes Parsen des kompletten Build-Servers eingeleitet. Durch diesen Vorgang werden alle neuen Versionen der Services vom Build-Server in die Datenbank übertragen, sowie eventuelle Abhängigkeiten erzeugt.

7. Fazit

Obwohl seit längerer Zeit Services und Serviceorientierte Architekturen aus der Welt der Informatik nicht mehr wegzudenken sind, scheint die Vereinfachung der Installation von Services keine hohe Priorität zu genießen. So ist es bemerkenswert, dass es wenige Programme gibt, die sich der Automatisierung der Installation annehmen. Die wenigen Programme, die es gibt, spezialisieren sich meist auf bestimmte Teilgebiete der Installation, lassen aber oft eine Automatisierung des gesamten Vorgangs vermissen. Die Auflösung von Abhängigkeiten scheint hier eine besonders große Herausforderung zu sein, da keines der betrachteten Werkzeuge hier eine Unterstützung liefert. Dabei kann es gerade bei wachsender Zahl von Services schnell zu vielen Abhängigkeiten kommen, die manuell kaum noch zu bewältigen sind.

Selbst wenn man die Abhängigkeiten unberücksichtigt lässt, ist die Unterstützung für immer wiederkehrende und sich kaum zu unterscheidende Aufgaben nicht besonders gut. So muss man sich fragen, warum die JavaEE Spezifikation hier nicht schon einen gangbaren Weg zur automatisierten Installation von Komponenten vorsieht.

Durch die Einführung der EJB 3.0 Spezifikation wurden immerhin schon viele der Konfigurationsschritte des Deployment Descriptors vereinfacht. Dies scheint ein erster Schritt in die richtige Richtung zu sein, löst das Problem aber noch nicht vollständig. Noch immer wird eine Person benötigt, welche Ressourcen anlegt und alle benötigten Services von Hand installiert.

Durch den entwickelten Prototypen wurde aufgezeigt, dass eine Automatisierung in dem Bereich der Installation grundsätzlich möglich ist. Hierbei wurde, soweit möglich, auf bereits bestehende Spezifikationen zurückgegriffen. Durch das offene und erweiterbare Konzept ist es möglich, auch andere Applikationsserver zu unterstützen. Vor allem das Auflösen der Abhängigkeiten, das Anlegen von Ressourcen und die Möglichkeit der Anpassung von Konfigurationsdateien erleichtern die Installation von Services erheblich.

In Zukunft wird sich zeigen müssen, ob durch die JavaEE Spezifikation hier eine Lösung vorgeben wird oder weiterhin Insellösungen durch spezielle Programme entstehen werden.

8. Literaturverzeichnis

EJB3 Ihns, Oliver et al.; EJB 3 professional; 2007

EinstiegXML Vonhoegen, Helmut; Einstieg in XML; 5. Auflage 2009

JAXB2.0 Michaelis, Samuel; Schmiesing, Wolfgang; JAXB 2.0; 2007

JavaInsel Ullenboom, Christian; Java ist auch eine Insel; 6. Auflage 2007

JavaEE5Tut Jendrock, Eric et al.; The Java EE 5 Tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc/ (Stand 13.04.2010)

JSR88Spec Dochez, Jerome; JavaTM Enterprise Edition 5 Deployment API Specification, Version 1.2, http://jcp.org/aboutJava/communityprocess/mrel/jsr088/index.html (Stand 25.11.2009)

JavaEE5API o.V.; Java™ Platform Enterprise Edition, v 5.0 API Specifications, http://java.sun.com/javaee/5/docs/api/ (Stand 5.12.2009

Java5API o.V.; Java™ 2 Platform Standard Edition 5.0 API Specification, http://java.sun.com/j2se/1.5.0/docs/api/ (Stand 5.12.2009)

JSR220Spec o.V.; JSR220: Enterprise JavaBeans, Version 3.0, http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html (Stand 15.4.2010)

GFishDeplGuide o.V.; Sun GlassFish Enterprise Server 2.1 Application Deployment Guide, http://dlc.sun.com/pdf/820-4337/820-4337.pdf (Stand 6.4.2010)

GFAdminGuide o.V.; Sun GlassFish Enterprise Server 2.1 Administration Guide, http://dlc.sun.com/pdf/820-4335/820-4335.pdf (Stand 6.4.2010)

JAXBTut Laun, Wolfgang; A JAXB Tutorial, https://jaxb.dev.java.net/tutorial/ (Stand 20.1.2010)

JAXBAPI o.V.; JAXB 2.2 Runtime Library, https://jaxb.dev.java.net/nonav/2.2-ea/docs/api/ (Stand 20.1.2010)

JAXBGuide o.V.; Unofficial JAXB Guide, https://jaxb.dev.java.net/guide/ (Stand 20.1.2010)

HudsonUse o.V.; Use Hudson (Manual), http://wiki.hudson-ci.org/display/HUDSON/Use+Hudson (Stand 13.12.2009)

JMXSpec o.V.; Java Managment Extension Specification, version 1.4, http://jcp.org/aboutJava/communityprocess/mrel/jsr003/index3.html (Stand 19.12.2009)

XPath2.0 o.V.;XML Path Language (XPath) 2.0, http://www.w3.org/TR/xpath20/ (Stand 17.12.2009)

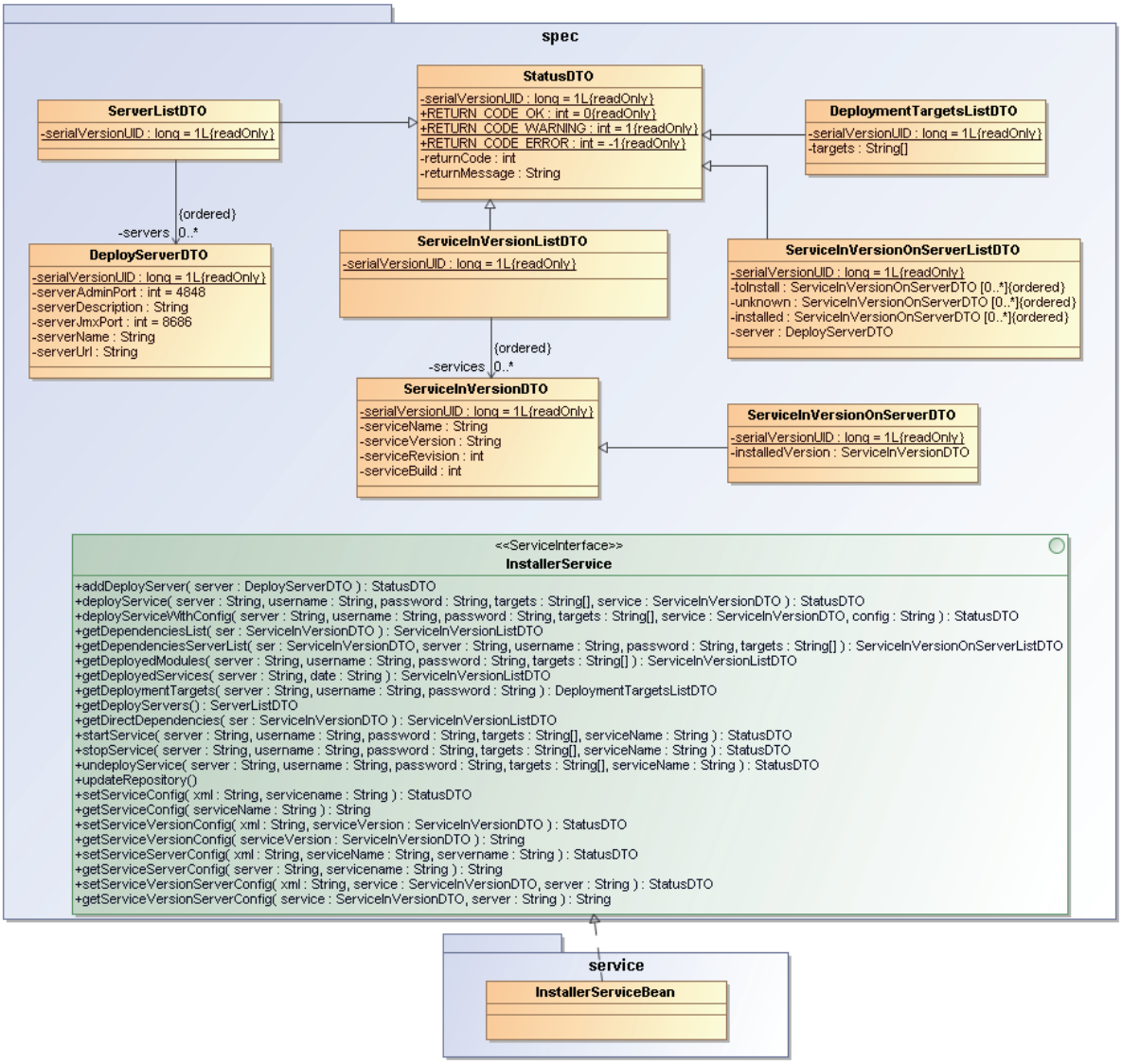
WikiUML o.V.; Unified Modeling Language, http://de.wikipedia.org/wiki/Unified_Modeling_Language#Metamodellierung (Stand 10.12.2009)

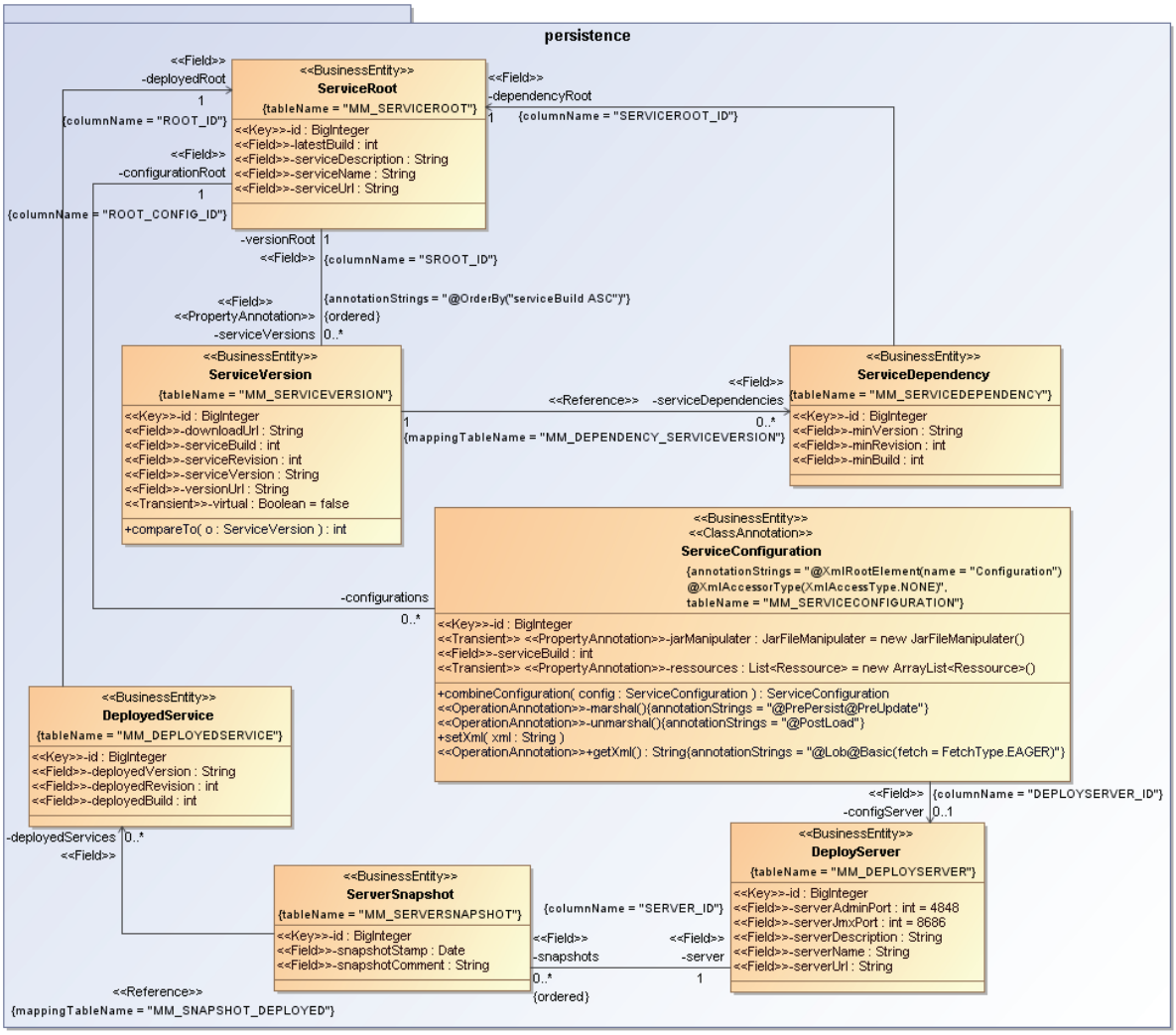
DPattern04 Freemann, Eric et al.; Entwurfsmuster von Kopf bis Fuß; 4. Auflage 2008

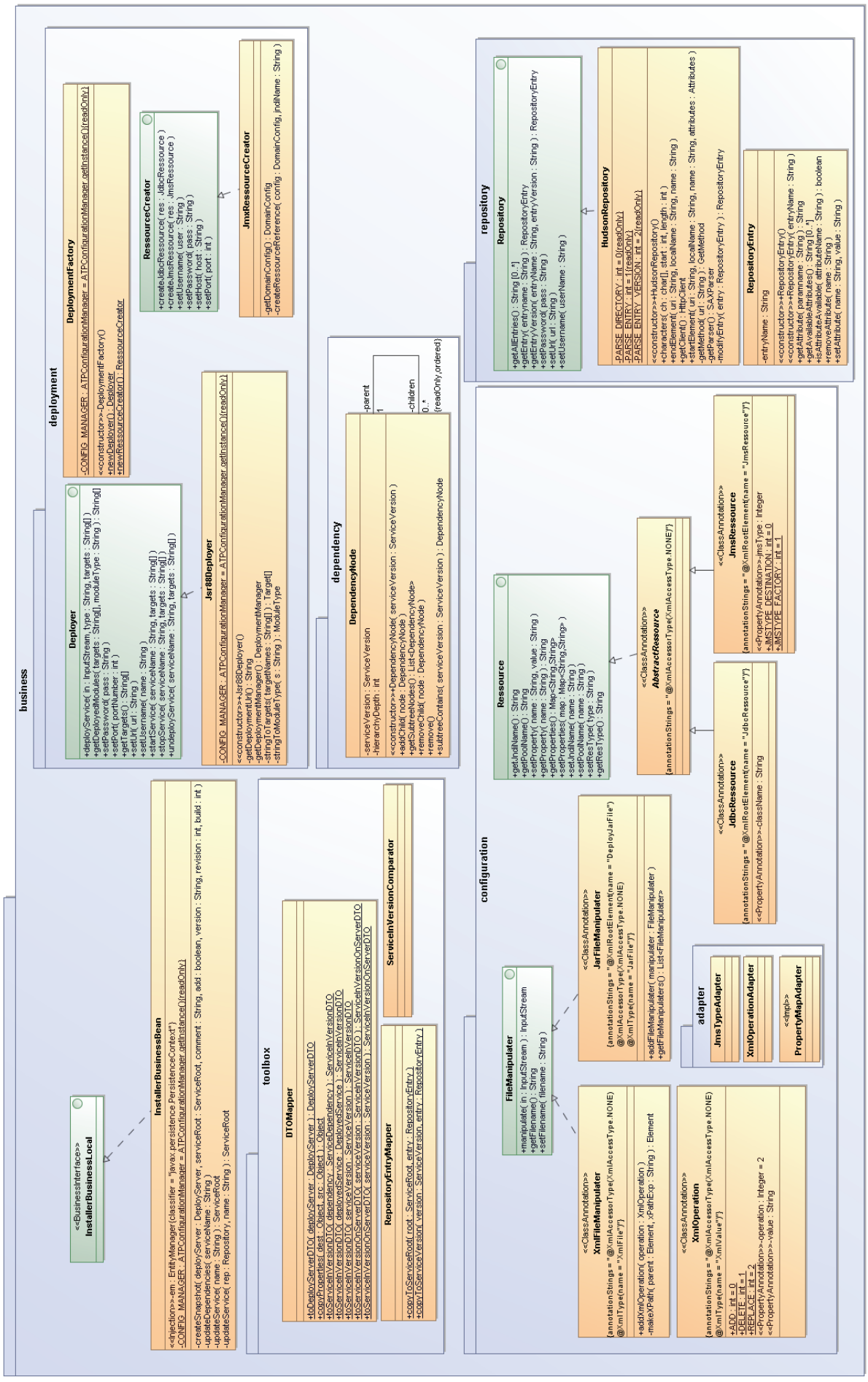
9. Anhang

9.1. Klassendiagramme

Auf den folgenden Seiten wird das komplette Klassendiagramm des implementierten Prototyps dargestellt.




persistence
<<BusinessEntity>>
ServiceRoot
{tableName = "MM_SERVICEROOT"}
<<Key>>-id : BigInteger
<<Field>>-latestBuild : int
<<Field>>-serviceDescription : String
<<Field>>-serviceName : String
<<Field>>-serviceUrl : String
<<Field>> -deployedRoot 1 {columnName = "ROOT_ID"}
<<Field>> -configurationRoot 1 {columnName = "ROOT_CONFIG_ID"}
<<Field>> -dependencyRoot 1 {columnName = "SERVICEROOT_ID"}
-versionRoot 1 <<Field>> {columnName = "SROOT_ID"}
<<Field>> <<PropertyAnnotation>> -serviceVersions 0..* {annotationStrings = "@OrderBy("serviceBuild ASC")"} {ordered}
<<BusinessEntity>>
ServiceVersion
{tableName = "MM_SERVICEVERSION"}
<<Key>>-id : BigInteger
<<Field>>-downloadUrl : String
<<Field>>-serviceBuild : int
<<Field>>-serviceRevision : int
<<Field>>-serviceVersion : String
<<Field>>-versionUrl : String
<<Transient>>-virtual : Boolean = false
+compareTo(o : ServiceVersion) : int
<<Reference>> 1 {mappingTableName = "MM_DEPENDENCY_SERVICEVERSION"}
<<Field>> -serviceDependencies 0..*
<<BusinessEntity>>
ServiceDependency
{tableName = "MM_SERVICEDEPENDENCY"}
<<Key>>-id : BigInteger
<<Field>>-minVersion : String
<<Field>>-minRevision : int
<<Field>>-minBuild : int
<<BusinessEntity>>
<<ClassAnnotation>>
ServiceConfiguration
{annotationStrings = "@XmlRootElement(name = "Configuration")
@XmlAccessorType(XmlAccessType.NONE)",
tableName = "MM_SERVICECONFIGURATION"}
<<Key>>-id : BigInteger
<<Transient>> <<PropertyAnnotation>>-jarManipulater : JarFileManipulater = new JarFileManipulater()
<<Field>>-serviceBuild : int
<<Transient>> <<PropertyAnnotation>>-ressources : List<Ressource> = new ArrayList<Ressource>()
+combineConfiguration(config : ServiceConfiguration) : ServiceConfiguration
<<OperationAnnotation>>-marshal(){annotationStrings = "@PrePersist@PreUpdate"}
<<OperationAnnotation>>-unmarshal(){annotationStrings = "@PostLoad"}
+setXml(xml : String)
<<OperationAnnotation>>+getXml() : String{annotationStrings = "@Lob@Basic(fetch = FetchType.EAGER)"}
-configurations 0..*
<<Field>> {columnName = "DEPLOYSERVER_ID"} -configServer 0..1
<<BusinessEntity>>
DeployedService
{tableName = "MM_DEPLOYEDSERVICE"}
<<Key>>-id : BigInteger
<<Field>>-deployedVersion : String
<<Field>>-deployedRevision : int
<<Field>>-deployedBuild : int
-deployedServices 0..* <<Field>>
<<BusinessEntity>>
DeployServer
{tableName = "MM_DEPLOYSERVER"}
<<Key>>-id : BigInteger
<<Field>>-serverAdminPort : int = 4848
<<Field>>-serverJmxPort : int = 8686
<<Field>>-serverDescription : String
<<Field>>-serverName : String
<<Field>>-serverUrl : String
<<BusinessEntity>>
ServerSnapshot
{tableName = "MM_SERVERSNAPSHOT"}
<<Key>>-id : BigInteger
<<Field>>-snapshotStamp : Date
<<Field>>-snapshotComment : String
{columnName = "SERVER_ID"}
<<Field>> -snapshots 0..* {ordered}
<<Field>> -server 1
<<Reference>> {mappingTableName = "MM_SNAPSHOT_DEPLOYED"}


business
deployment
DeploymentFactory
-CONFIG_MANAGER : ATPConfiguratorManager = ATPConfiguratorManager.getInstance()(readOnly)
<<constructor>>-DeploymentFactory()
+newDeployer() : Deployer
+newRessourceCreator() : RessourceCreator
Deployer
+deployService(in : InputStream, type : String, targets : String[])
+getDeployedModules(targets : String[], moduleType : String) : String[]
+setHost(hostAddress : String)
+setPort(portNumber : int)
+getTargets() : String[]
+setUrl(url : String)
+setUsername(name : String)
+startService(serviceName : String, targets : String[])
+stopService(serviceName : String, targets : String[])
+undeployService(serviceName : String, targets : String[])
RessourceCreator
+createJdbcRessource(res : JdbcRessource)
+createJmsRessource(res : JmsRessource)
+setUsername(user : String)
+setPassword(pass : String)
+setHost(host : String)
+setPort(port : int)
Jsr88Deployer
-CONFIG_MANAGER : ATPConfiguratorManager = ATPConfiguratorManager.getInstance()(readOnly)
<<constructor>>+Jsr88Deployer()
-getDeploymentManager() : DeploymentManager
-stringToTargets(targetNames : String[]) : Target[]
-stringToModuleType(s : String) : ModuleType
JmxRessourceCreator
-getDomainConfig() : DomainConfig
-createRessourceReference(config : DomainConfig, jndiName : String)
dependency
DependencyNode
-serviceVersion : ServiceVersion
-hierarchyDepth : int
<<constructor>>+DependencyNode(serviceVersion : ServiceVersion)
+addChild(node : DependencyNode)
+getChildren() : List<DependencyNode>
+removeChild(node : DependencyNode)
+remove()
+subTreeContains(serviceVersion : ServiceVersion) : DependencyNode
-parent
1
-children
0..*
{readOnly,ordered}
repository
Repository
+getAllEntries() : String [0..*]
+getEntry(entryName : String) : RepositoryEntry
+getEntryVersion(entryName : String, entryVersion : String) : RepositoryEntry
+setPassword(pass : String)
+setUrl(url : String)
+setUsername(userName : String)
HudsonRepository
-PARSE_DIRECTORY : int = 0{readOnly}
-PARSE_ENTRY : int = 1{readOnly}
-PARSE_ENTRY_VERSION : int = 2{readOnly}
<<constructor>>+HudsonRepository()
+characters(ch : char[], start : int, length : int)
+endElement(uri : String, localName : String, name : String)
+getClient() : HttpClient
+startElement(uri : String, localName : String, name : String, attributes : Attributes)
-getMethod(url : String) : GetMethod
-getParser() : SAXParser
-modifyEntry(entry : RepositoryEntry) : RepositoryEntry
RepositoryEntry
-entryName : String
<<constructor>>+RepositoryEntry()
<<constructor>>+RepositoryEntry(entryName : String)
+addAttribute(name : String, value : String)
+getAvailableAttributes() : String [0..*]
+isAttributeAvailable(attributeName : String) : boolean
+removeAttribute(name : String)
+setAttribute(name : String, value : String)
configuration
Ressource
+getJndiName() : String
+getName() : String
+setProperty(name : String, value : String)
+getProperty(name : String) : String
+getProperties() : Map<String,String>
+setProperties(map : Map<String,String>)
+setJndiName(name : String)
+setPoolName(name : String)
+getResType(type : String)
+setResType() : String
<<ClassAnnotation>>
AbstractRessource
{annotationStrings = "@XmlAccessorType(XmlAccessType.NONE)"}
<<ClassAnnotation>>
JdbcRessource
{annotationStrings = "@XmlRootElement(name = "JdbcRessource")"}
<<PropertyAnnotation>>-className : String
<<ClassAnnotation>>
JmsRessource
{annotationStrings = "@XmlRootElement(name = "JmsRessource")"}
<<PropertyAnnotation>>-jmsType : Integer
+JMSTYPE_DESTINATION : int = 0
+JMSTYPE_FACTORY : int = 1
FileManipulater
+manipulate(in : InputStream) : InputStream
+getFilename() : String
+setFilename(filename : String)
<<ClassAnnotation>>
JarFileManipulater
{annotationStrings = "@XmlRootElement(name = "DeployJarFile")
@XmlAccessorType(XmlAccessType.NONE)
@XmlType(name = "JarFile")"}
+addFileManipulater(manipulater : FileManipulater)
+getFileManipulaters() : List<FileManipulater>
adapter
JmsTypeAdapter
XmlOperationAdapter
<<Impl>>
PropertyMapAdapter
<<ClassAnnotation>>
XmlFileManipulater
{annotationStrings = "@XmlAccessorType(XmlAccessType.NONE)
@XmlType(name = "XmlFile")"}
+addXmlOperation(operation : XmlOperation)
-makeXPath(parent : Element, xPathExp : String) : Element
<<ClassAnnotation>>
XmlOperation
{annotationStrings = "@XmlAccessorType(XmlAccessType.NONE)
@XmlType(name = "XmlValue")"}
+ADD : int = 0
+DELETE : int = 1
+REPLACE : int = 2
<<PropertyAnnotation>>-operation : Integer = 2
<<PropertyAnnotation>>-value : String
toolbox
DTOMapper
+toDeployServerDTO(deployServer : DeployServer) : DeployServerDTO
+copyProperties(dest : Object, orig : Object)
+toServiceInVersionDTO(dependency : ServiceDependency) : ServiceInVersionDTO
+toDeployServerDTO(deployedService : DeployedService) : ServiceInVersionDTO
+toServiceInVersionDTO(serviceVersion : ServiceVersion) : ServiceInVersionDTO
+toServiceInVersionOnServerDTO(serviceVersion : ServiceInVersionDTO) : ServiceInVersionOnServerDTO
+toServiceInVersionOnServerDTO(serviceVersion : ServiceVersion) : ServiceInVersionOnServerDTO
ServiceInVersionComparator
RepositoryEntryMapper
+copyToServiceRoot(root : ServiceRoot, entry : RepositoryEntry)
+copyToServiceVersion(version : ServiceVersion, entry : RepositoryEntry)
installerBusinessBean
<<Injection>>-em : EntityManager(classifier = "javax.persistence.PersistenceContext")
-CONFIG_MANAGER : ATPConfiguratorManager = ATPConfiguratorManager.getInstance()(readOnly)
-createSnapshot(deployServer : DeployServer, serviceRoot : ServiceRoot, comment : String, add : boolean, version : String, revision : int, build : int)
+createService(name : String) : ServiceRoot
-updateService(name : String) : ServiceRoot
-updateService(rep : Repository, name : String) : ServiceRoot
<<BusinessInterface>>
InstallerBusinessLocal

Weitere Anhänge befinden sich in elektronischer Form auf der beigelegten CD.

- Diplomarbeit als Word-Dokument
- Diplomarbeit als PDF-Datei
- Abbildungen im PNG-Format
- Quelltexte des Prototyps

Erklärung über die selbstständige Abfassung der Arbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt, bzw. in wesentlichen Teilen, noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, den 6. Mai 2010

(Markus-A. Müller)